Comparison of Two Different Control Software for Path Following: Subsumption Architecture vs. Motor Schemas

Daniele Commodaro
daniele.commodaro@studio.unibo.it

July 2024

Path following is a fundamental task in robotic systems, essential for applications that deal with realizing autonomous and resilient behaviors. This project compares two distinct control architectures, Subsumption and Motor Schemas, highlighting their design principles, implementation details, and performance in the tasks that follow the path. The goal is to identify the strengths and weaknesses of each approach to better understand its potential and perhaps to guide possible future developments in intelligent systems engineering.

Contents

1 Requirements

The project requires implementing a behavior-based controller for both Subsumption and Motor Schemas architectures. Each architecture must support and combine several types of behaviors, including:

1. Phototaxis mechanism: a behavior guiding the robot towards a light source.
2. Obstacle detection and avoidance mechanism: a behavior enabling the robot to detect and avoid obstacles in its path.
3. Random movement mechanism: a behavior causing the robot to move randomly when no light source is detected.

Additionally, the project requires an experimental analysis, involving the following steps:

- Objective definition and hypothesis formulation: setting clear goals for the experiments and formulating hypotheses regarding the expected outcomes.
- Analysis setup: designing the experiments to test the behaviors and overall performance of each architecture.
- Result analysis: collecting and analyzing the data obtained from the experiments using statistical methods and graphical representations.
- Hypothesis evaluation: assessing the validity of the hypotheses based on the experimental results.

By meeting these requirements, the project aims to provide a thorough comparison of Subsumption and Motor Schemas architectures in terms of their effectiveness in path-following tasks.

2 Architectural design

2.1 Subsumption architecture

The Subsumption Architecture is a behavior-based control strategy that organizes robot behaviors in a layered manner. Each layer corresponds to a specific behavior with higher-priority behaviors capable of inhibiting or subsuming lower-priority ones.

2.1.1 State of art

The Subsumption Architecture represents a significant departure from traditional AI and robotics control systems. It emphasizes behavior-based control, eschewing centralized planning in favor of decentralized, reactive behaviors. This architecture has been highly influential, forming the basis for many modern autonomous robotic systems.

The core idea of the Subsumption Architecture is to break down complex robotic tasks into simpler, hierarchical layers of behaviors. Each layer is responsible for a specific behavior and can subsume the actions of lower-priority layers if certain conditions are met. This layered approach enables robots to exhibit sophisticated behaviors through the interaction of simpler modules.

- Behavioral decomposition: the architecture decomposes robotic tasks into discrete behaviors, such as obstacle avoidance, phototaxis, and wall following. Each behavior operates independently, processing sensor data and generating actuator commands.
- Layered control: behaviors are organized in a priority hierarchy. Lower layers handle basic reactive behaviors, while higher layers deal with more complex, goal-oriented behaviors.

- Asynchronous execution: each behavior operates asynchronously, reacting to sensor inputs without waiting for a centralized controller. This allows for rapid responses to environmental changes, enhancing the robot's robustness and adaptability.
- Suppression and inhibition: higher-priority behaviors can suppress or inhibit lower-priority ones. For example, an obstacle avoidance behavior might override the phototaxis behavior if an obstacle is detected, ensuring the robot avoids collisions while still pursuing its goal.

The Subsumption Architecture remains a foundational approach in robotics, offering a robust framework for developing autonomous systems capable of operating in dynamic environments. Its influence extends beyond robotics, informing the design of distributed control systems and inspiring research into behavior-based AI. As technology advances, ongoing research continues to refine and extend this architecture, ensuring its relevance in the evolving field of intelligent systems engineering.

2.1.2 Behaviors

The design of this kind of controller involves several key behaviors and their interactions:

1. Phototaxis behavior: this behavior directs the robot towards a light source. It calculates the resultant light vector by summing the readings from all light sensors. If a light source is detected the robot turns and moves towards it. This behavior counts the time spent without any light source detection, when this time reaches a certain threshold a random movement behavior is triggered
2. Obstacle avoidance behavior: this behavior uses proximity sensors to detect obstacles and calculate a resultant vector for avoidance. If the obstacle detection exceeds a proximity threshold, the robot adjusts its path to avoid the obstacle.
3. Random movement behavior: when no light source is detected for a specified number of steps, the robot enters a random movement mode. In this mode, the robot moves randomly, adjusting its direction and speed periodically to explore the environment.
4. Wall following behavior: if the robot detects a wall using its proximity sensors, it follows the wall by adjusting its movement to maintain a certain distance from the wall.

2.1.3 Behaviors prioritization

In this architecture, behaviors are prioritized to ensure appropriate responses to environmental stimuli:

- Obstacle avoidance: has the highest priority, ensuring that the robot does not collide with obstacles.
- Wall following: is the next highest priority, activated when the robot encounters walls along his path. When this behavior is executed it can also take into account any brightness detections so as not to ignore them while continuing to follow the wall. This logic was introduced to not ignore the final destination while following a wall, in order to try to obtain behavior that is as realistic as possible.
- Phototaxis: follows, guiding the robot towards a light source when no obstacles or walls are detected.
- Random movement: has the lowest priority and is only activated when no light source is detected for a prolonged period.

2.1.4 Control loop

The main control loop orchestrates the execution of behaviors in sequence. Each behavior, when executed, returns a rotation angle based on the data collected from the sensors. After executing the behaviors, the control loop invokes the actuator method, which uses the angle value returned by one of the behaviors to orient the robot towards that angle. The decision on which angle to use is based on the established priority levels of the various behaviors. If none of the behaviors return relevant values, the robot simply moves straight. If no light is detected for an extended period, the random movement behavior is activated.

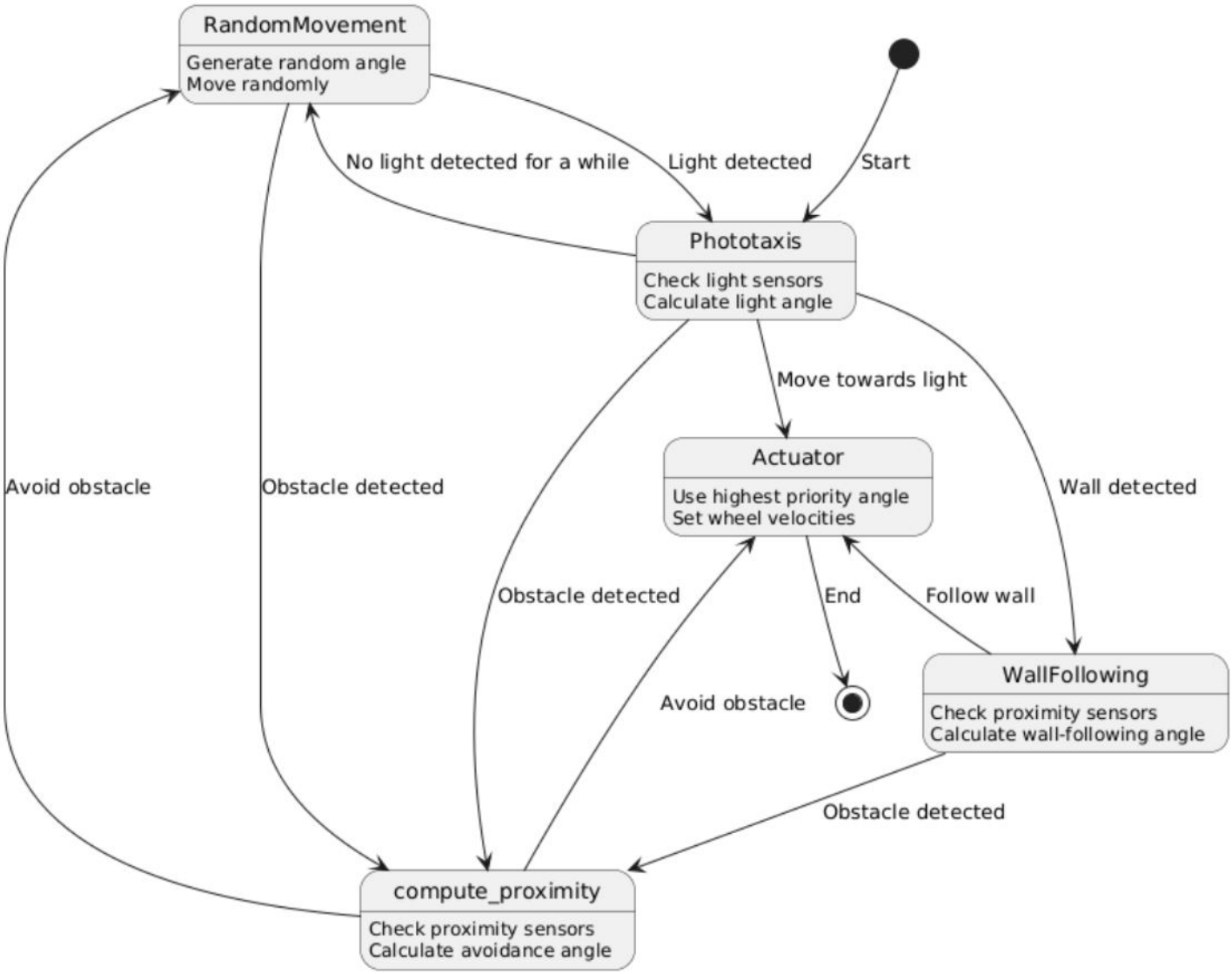

Figure 1: FSM of the Subsumption architecture

The FSM diagram illustrates the different behaviors that make up the robot controller and the actions that are performed in the form of transitions. Behaviors are prioritized and executed based on data sensed through sensors and their hierarchical importance, ensuring that the robot responds appropriately to environmental stimuli.

2.2 Motor Schemas

2.2.1 State of art

Motor Schemas represent a powerful paradigm in the field of autonomous robotics, particularly for the coordination of multiple behaviors. Inspired by biological systems, Motor Schemas offer a way to achieve complex behavior through the combination of simpler, individual actions. This approach stands out for its ability to produce robust and adaptable robotic behaviors in dynamic environments.

Motor Schemas are a method of behavior-based control where each behavior generates a vector that represents a desired movement direction and magnitude. These vectors are then combined to produce a resultant vector that dictates the robot's movement. Key aspects include:

- Vector-based behavior representation: each behavior, or schema, outputs a vector. The direction and magnitude of these vectors represent the intended movement of the robot based on sensor inputs.
- Weighted vector summation: to combine multiple behaviors, vectors are weighted according to the importance or priority of the corresponding behavior. The weighted vectors are then summed to produce a final resultant vector that guides the robot's movement.
- Reactive and adaptive: this approach allows robots to react quickly to changes in the environment by adjusting the contributions of different behaviors in real-time. It also supports the blending of behaviors to achieve smooth transitions and more nuanced responses.

2.2.2 Behaviors and architecture

The Motor Schemas architecture for robots consists of several key behaviors:

1. Phototaxis: this behavior directs the robot towards a light source. The robot's light sensors detect the intensity and direction of light, and the behavior generates a vector pointing towards the light source. If the light intensity is below a threshold for a certain number of steps, the robot will switch to a random movement mode to explore the environment.
2. Obstacle avoidance: this behavior prevents the robot from colliding with obstacles. Using proximity sensors, it generates a vector pointing away from detected obstacles. This vector is weighted heavily to ensure the robot prioritizes avoiding collisions.
3. Wall following: this behavior allows the robot to navigate along walls or boundaries. It identifies the strongest proximity sensor reading and generates a vector that keeps the robot parallel to the detected wall. This helps the robot navigate enclosed spaces effectively.
4. Random movement: when no significant light source is detected for an extended period, the robot engages in random movement to explore its surroundings. This behavior generates random vectors to prevent the robot from getting stuck in one place. During this type of behavior, the vectors that determine the movement are combined with the vectors of the behavior that deal with obstacle avoidance, to try to give the movement an appearance as realistic as possible.

The main control loop executes these behaviors in sequence at each time step:

- Phototaxis: computes a vector based on light sensor data. If the light intensity is above the threshold, it resets the step counter and exits random movement mode if it is active.
- Obstacle avoidance: computes a vector based on proximity sensor data from the front sensors, creating a repulsive force away from obstacles.

- Wall following: computes a vector to follow walls using the proximity sensors, ensuring the robot navigates effectively in confined spaces.
- Random Movement: generates a random vector if the robot has not detected significant light for a specified number of steps.

The resultant movement vector is computed by summing the weighted vectors from each behavior. The weights determine the influence of each behavior on the final movement direction. Obstacle avoidance has a greater weight to ensure correct behavior, the second behavior with greater weight is that for wall following, after which there is the phototaxis behavior to direct the robot towards light sources, and finally the random behavior.

Also in this case there is an actuator function that uses the resulting vector to set the speed of the robot's wheels, translating the direction and magnitude of the vector into physical movement. If no significant resultant vector is found, the robot moves straight.

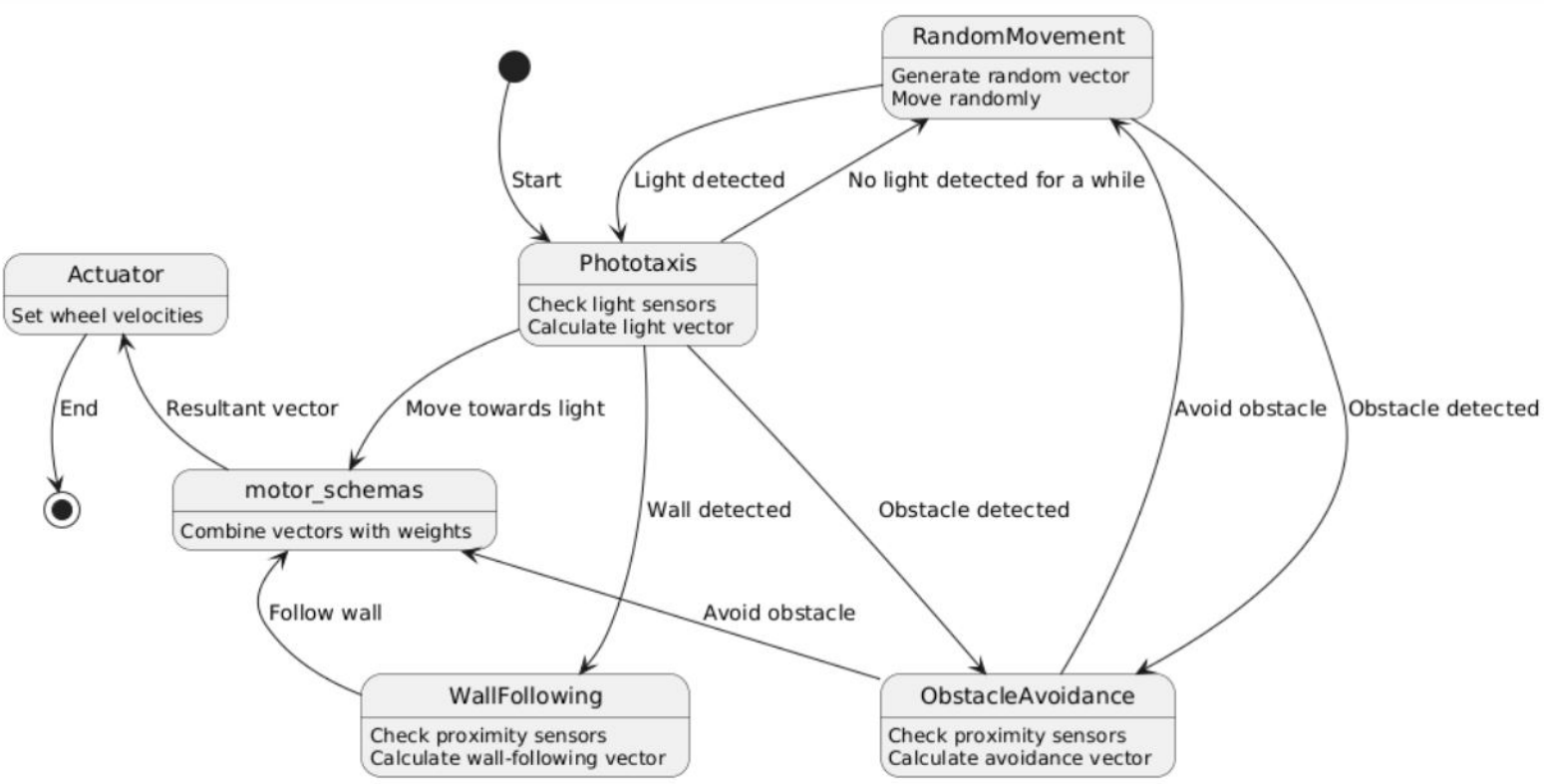

Figure 2: FSM of the Motor Schemas architecture

This architecture offers a robust, flexible framework for autonomous robotic control. By combining multiple behavior vectors through weighted summation, it allows for the creation of complex, adaptive behaviors that enable the robot to navigate and interact with its environment effectively.

3 Detailed design

3.1 Subsumption architecture

In this implementation, a modular approach is employed where different behaviors compete for control of the robot. The various behaviors include phototaxis, obstacle avoidance, wall-following, and random movement, which are combined through a series of priority logics.

Global variables

The following global variables define the thresholds and store the state of the robot:

```
local LIGHT_THRESHOLD = 0
local NO_LIGHT_STEPS_THRESHOLD = 100
local PROXIMITY_THRESHOLD = 0.25

local light_angle = 0
local step_counter = 0
local random_movement_mode = false
local random_movement_step_counter = 0
local proximity_avoidance_angle = 0
local proximity_resultant_length = 0
local wall_following_vector = {length = 0, angle = 0}
```

LIGHT_THRESHOLD is the threshold beyond which light detection is considered relevant. Similarly, PROXIMITY_THRESHOLD is the proximity threshold beyond which the detection of an obstacle is taken into account. NO_LIGHT_STEPS_THRESHOLD, on the other hand, is the threshold that defines the number of steps without detecting light after which random movement should be activated.

The variables are used to store the robot's state and consist of the following data:

- light_angle: stores the resulting angle from the sum of the vectors given by light detection through the sensors.
- step_counter: stores the number of steps during which no light is detected.
- random_movement_mode: indicates whether random movement is active.
- random_movement_step_counter: stores the number of steps counted during the execution of random movement.
- proximity_avoidance_angle: stores the angle detected between the robot and an obstacle.
- proximity_resultant_length: stores the value of the vector returned by the proximity sensors for obstacle detection.
- wall_following_vector: stores the vector returned by the proximity sensors for wall detection.

Phototaxis

The phototaxis behavior reads the robot's light sensor data, calculates a direction for the robot to move towards the light, and sets the robot's movement mode based on these readings.

```
local light_readings = robot.light
local cAccumulatorX = 0
local cAccumulatorY = 0
```

The variable light_readings stores the robot's light sensor values while the two accumulators cAccumulatorX and cAccumulatorY will be used to calculate the light direction in cartesian coordinates.

```
for i = 1, #light_readings do
   local sensor_value = light_readings[i].value
   local sensor_angle = light_readings[i].angle
   cAccumulatorX = cAccumulatorX + sensor_value * math.cos(sensor_angle)
   cAccumulatorY = cAccumulatorY + sensor_value * math.sin(sensor_angle)
end
```

The for loop iterates through all the robot's light sensors, and from each one it obtains the intensity value and the angle detected.

The accumulators cAccumulatorX and cAccumulatorY are updated by summing the contribution of each sensor, which is calculated using the appropriate trigonometric function (cosine for the X component and sine for the Y component). Sine and cosine are used to convert the angular and intensity information relative to the light intensity values into cartesian coordinates. This allows all sensor readings to be summed into a single resultant direction. Thus, sensor_value * math.cos(sensor_angle) gives the X component of the sensor's contribution, while sensor_value * math.sin(sensor_angle) gives the Y component of the sensor's contribution.

```
local resultant_angle = math.atan2(cAccumulatorY, cAccumulatorX)
local resultant_length = math.sqrt(cAccumulatorX^2 + cAccumulatorY^2)
```

Then, with math.atan2(cAccumulatorY, cAccumulatorX), the resulting angle of the light direction is calculated using the atan2 function, which returns the angle of the resultant vector in polar coordinates. With math.sqrt($cAccumulatorX^2 + cAccumulatorY^2$), the length of the resultant vector is calculated, representing the overall strength of the perceived light.

```
if resultant_length > LIGHT_THRESHOLD then
   light_angle = resultant_angle
   step_counter = 0
   random_movement_mode = false
else
   light_angle = 0
   step_counter = step_counter + 1
   if step_counter > NO_LIGHT_STEPS_THRESHOLD then
   random_movement_mode = true
   end
end
```

In the end, it is determined whether the overall detected light intensity is greater than a threshold LIGHT_THRESHOLD. If so, the robot will subsequently use the resulting angle resultant_angle to orient itself towards the detected light.

If the light intensity is below the set threshold, the robot increments a counter step_counter. If this counter exceeds the NO_LIGHT_STEPS_THRESHOLD, the robot sets a flag through which it will subsequently enter a random movement mode.

Movement along walls

The wall following behavior reads the robot's proximity sensor values to find the sensor detecting the closest obstacle, calculates a movement vector to help the robot follow the wall, and returns this vector.

```
local vtr = {length = 0, angle = 0}
local value = -1
local idx = -1
```

The variables used for this behavior are: vtr, which is a vector containing the length and angle of the desired movement. value is a variable that keeps track of the maximum value detected by the proximity sensors, and idx is a variable that keeps track of the index of the sensor with the maximum value.

```
for i = 1, 24 do
   if value < robot.proximity[i].value then
      idx = i
      value = robot.proximity[i].value
   end
end
```

The for loop searches for the maximum value among the 24 proximity sensors. In the loop, all the robot's sensors are iterated through, and if the value read by a sensor is greater than the current maximum detected value, then the index and value of this sensor are updated.

```
if idx >= 0 then
   local angle = robot.proximity[idx].angle
   if angle <= 0 then
      vtr.angle = angle + math.pi / 2
      vtr.length = value
   else
      vtr.angle = angle - math.pi / 2
      vtr.length = value
   end
else
   vtr.length = 0
   vtr.angle = 0
end
```

The selection construct is responsible for calculating the necessary movement vector. The angle of the sensor with the maximum value is obtained with the command: robot.proximity[idx].angle. If the angle is less than or equal to 0, a new angle is calculated for the robot's rotation of 90 degrees to the left plus the value of the detected angle, otherwise if the angle is greater than 0, a rotation angle of 90 degrees is calculated to the right added to the value of the detected angle. The calculated rotation angle is returned along with the value of the maximum vector found. If no sensor has detected a significant obstacle, the returned vector values will be zero.

Obstacle avoidance

This behavior processes the robot's front proximity sensor data to calculate a resultant vector that represents the direction and strength of nearby obstacles.

```
local proximity_readings = robot.proximity
local cAccumulatorX = 0
local cAccumulatorY = 0
```

The behavior begins by initializing some variables: proximity_readings, initialized with the robot's proximity sensor values, and two accumulators, cAccumulatorX and cAccumulatorY, initialized to zero. These accumulators will be used to calculate the resultant direction of the obstacles.

```
for i = 1, 6 do
   local sensor_value = proximity_readings[i].value
   local sensor_angle = proximity_readings[i].angle
   cAccumulatorX = cAccumulatorX + sensor_value * math.cos(sensor_angle)
   cAccumulatorY = cAccumulatorY + sensor_value * math.sin(sensor_angle)
end
for i = 19, 24 do
   local sensor_value = proximity_readings[i].value
   local sensor_angle = proximity_readings[i].angle
   cAccumulatorX = cAccumulatorX + sensor_value * math.cos(sensor_angle)
   cAccumulatorY = cAccumulatorY + sensor_value * math.sin(sensor_angle)
end
```

To avoid obstacles, only the front sensors are used, these sensors are in the ranges (1, 6) and (19, 24). Two for loops iterate over the robot's front sensors, reading the value of each returned vector and its angle. The values read in polar coordinates are converted using trigonometric

functions into their respective cartesian coordinates and accumulated in the respective variables.

```
local resultant_angle = math.atan2(cAccumulatorY, cAccumulatorX)
local resultant_length = math.sqrt(cAccumulatorX^2 + cAccumulatorY^2)
```

The accumulation variables are then converted into polar coordinates to obtain the resulting vector. math.atan2(cAccumulatorY, cAccumulatorX) is used to calculate the resulting angle of the obstacle direction using the atan2 function, and math.sqrt($cAccumulatorX^2$ + $cAccumulatorY^2$) is used to calculate the length of the resulting vector, which represents the overall strength of the perceived obstacles.

```
if math.abs(resultant_length) > PROXIMITY_THRESHOLD then
   proximity_avoidance_angle = resultant_angle
   proximity_resultant_length = resultant_length
else
   proximity_avoidance_angle = 0
   proximity_resultant_length = 0
end
```

Then, a threshold is used to determine whether obstacle avoidance should be implemented or not. If the length of the resulting vector is greater than the set threshold, the parameters of the resulting vector are stored and then returned, otherwise values set to zero will be returned.

Random movement

The random behavior controls the robot's movement by moving it randomly and potentially avoiding obstacles. If the robot detects an obstacle, it changes its trajectory by adjusting the wheel speeds to avoid it. If there are no obstacles, it generates a random angle and a number of steps to determine its movement. This process, once activated, repeats until a light source is detected.

```
if proximity_avoidance_angle ~= 0 then
   local avoidance_factor = math.min(proximity_resultant_length / PROXIMITY_THRESHOLD,
       1)
   local left_speed = 10 * (1 + 2 * avoidance_factor * proximity_avoidance_angle /
       math.pi)
   local right_speed = 10 * (1 - 2 * avoidance_factor * proximity_avoidance_angle /
       math.pi)
   robot.wheels.set_velocity(left_speed, right_speed)
   return
end
```

If an obstacle has been detected and thus proximity_avoidance_angle is not zero, an avoidance_factor is calculated, which depends on the proximity of the obstacle (the closer the obstacle, the greater the factor). Then, the wheel speeds are calculated based on the avoidance angle and the avoidance factor to turn the robot away from the obstacle, and then set with robot.wheels.set_velocity(left_speed, right_speed). Finally, the behavior exits by invoking return, since the obstacle needs to be avoided and the random behavior should not continue.

```
local random_angle = math.random() * 2 * math.pi - math.pi
local forward_steps = math.random(5, 20)
local curve_steps = math.random(5, 10)
```

The random parameters necessary for random movement are calculated: an angle ranging from -π to π, a number of steps to move forward, and a number of steps for turning.

```
if random_movement_step_counter < forward_steps then
   robot.wheels.set_velocity(10, 10)
```

```
elseif random_movement_step_counter < forward_steps + curve_steps then
   local left_speed = 10 * (1 - random_angle / 4)
   local right_speed = 10 * (1 + random_angle / 4)
   robot.wheels.set_velocity(left_speed, right_speed)
else
   random_movement_step_counter = 0
   return
end
random_movement_step_counter = random_movement_step_counter + 1
```

Finally, the execution of random movement is managed. If the number of random steps counted in random_movement_step_counter is less than the randomly generated forward_steps, the robot moves forward with equal velocity for both wheels. Once the expected number of forward steps is reached, the robot executes a turn by calculating the wheel velocities based on the generated random angle. Finally, if the total number of expected random steps is reached by the counter, it is reset, and the function exits using return, ready to update parameters for the next cycle.

Actuators control

The actuator decides the robot's movement behavior depending on the current state and the available sensory informations. If the robot is in random movement mode, it performs random movement, otherwise it chooses among different movement strategies based on the contextual conditions.

```
if random_movement_mode then
   random_movement()
```

The first selection statement shows that if the random movement mode has been activated during the phototaxis behavior, it will be executed, and all subsequent statements for the robot's movement will be ignored. In this case, the robot is in a state where it is not actually detecting light sources. The random movement manages any obstacle avoidance actions independently.

```
local wall_following_active = wall_following_vector.length > 0
local light_detected = math.abs(light_angle) > LIGHT_THRESHOLD
```

At the beginning of the actuator function, two local variables are initialized and will then be used to determine if the robot should follow a wall and to check if a light source has been detected.

```
if proximity_avoidance_angle ~= 0 then
   local avoidance_factor = math.min(proximity_resultant_length / PROXIMITY_THRESHOLD,
       1)
   local left_speed = 10 * (1 + 2 * avoidance_factor * proximity_avoidance_angle /
       math.pi)
   local right_speed = 10 * (1 - 2 * avoidance_factor * proximity_avoidance_angle /
       math.pi)
   robot.wheels.set_velocity(left_speed, right_speed)
```

In an order that reflects the priority within the behavior hierarchy, the results of various behaviors are applied based on whether they have returned relevant values or not. The first to be evaluated is the result of the highest-priority behavior, which is obstacle avoidance. If an obstacle has been detected and thus proximity_avoidance_angle is not zero, an avoidance factor based on the distance from the obstacle is calculated. The wheel velocities are then computed to avoid the obstacle by adjusting the direction according to the avoidance angle and factor, and finally, the wheel speeds are set. In calculating the wheel velocities, the division by π is used to normalize the angle value to a full rotation of 2π radians.

```
elseif wall_following_active and light_detected then
   local wall_following_speed = 10 * (1 - wall_following_vector.angle)
   local left_speed = wall_following_speed
   local right_speed = wall_following_speed

   left_speed = left_speed + 5 * (1 - light_angle)
   right_speed = right_speed + 5 * (1 + light_angle)

   robot.wheels.set_velocity(left_speed, right_speed)
```

If the obstacle avoidance behavior did not return values to apply, then it is evaluated whether a wall to avoid has been detected and if the robot is in a situation where a light source is being detected. In this case, if the robot is following a wall and detects the light, the wheel velocities are calculated based on the wall-following angle. To this calculated velocity, a component of velocity is added to move towards the light source. Finally, the wheel velocities are set, balancing the movement between following the wall and moving towards the light.

```
elseif wall_following_active then
   local left_speed = 10 * (1 - wall_following_vector.angle)
   local right_speed = 10 * (1 + wall_following_vector.angle)
   robot.wheels.set_velocity(left_speed, right_speed)
```

If the next statement is reached, it means that only a wall has been detected in the absence of light. In this case, the robot should simply follow the wall, so the wheel velocities are calculated to follow the wall based on the wall-following angle, and the wheel velocities are set to continue following the wall.

```
elseif light_detected then
   local left_speed = 10 * (1 - light_angle)
   local right_speed = 10 * (1 + light_angle)
   robot.wheels.set_velocity(left_speed, right_speed)
```

The second-to-last statement applies only the data for phototaxis movement. Therefore, if the robot detects only light, it calculates the wheel velocities to move towards the light based on the angle of the light, and then sets the wheel velocities to move towards the light source.

```
else
   robot.wheels.set_velocity(10, 10)
```

Finally, the last statement is reached only if none of the existing behaviors have returned relevant values. In this case, there are no obstacles, walls to follow, or detected light. Therefore, the robot is simply moved straight ahead with equal wheel velocities for both wheels.

Main controller

The main_controller() function is the main function that coordinates the execution of various behaviors of the robot in sequence. Each function call within main_controller() represents a specific behavior that the robot performs during its operation.

```
function main_controller()
   phototaxis()
   wall_following()
   compute_proximity()
   actuator()
end
```

The main_controller() function is in turn invoked by the step() method, which is executed at

each step of the simulation.

3.2 Motor Schemas

In this implementation, behaviors are represented as vectors that are summed together to determine the robot's movement. Each vector is associated with a weight that establishes the priority of one movement over another.

Constants and variables

```lua
local vector = require "vector"

local LIGHT_THRESHOLD = 0
local NO_LIGHT_STEPS_THRESHOLD = 100
local PROXIMITY_THRESHOLD = 0.2

local light_vector = {x = 0, y = 0}
local proximity_vector = {x = 0, y = 0}
local random_vector = {x = 0, y = 0}
local step_counter = 0
local random_movement_mode = false
local random_movement_step_counter = 0
```

In this first block, a library is imported to provide vector operations, and constants and global variables are defined to manage the robot's behaviors. The constants establish thresholds for light detection, the number of steps without light, and proximity to obstacles. The global variables store the movement vectors and count the steps.

Phototaxis

```lua
function phototaxis()
   local light_readings = robot.light
   local cAccumulatorX = 0
   local cAccumulatorY = 0

   for i = 1, #light_readings do
      local sensor_value = light_readings[i].value
      local sensor_angle = light_readings[i].angle
      cAccumulatorX = cAccumulatorX + sensor_value * math.cos(sensor_angle)
      cAccumulatorY = cAccumulatorY + sensor_value * math.sin(sensor_angle)
   end

   light_vector.x = cAccumulatorX
   light_vector.y = cAccumulatorY

   local resultant_length = vector.vec2_length({x = cAccumulatorX, y = cAccumulatorY})

   if resultant_length > LIGHT_THRESHOLD then
      step_counter = 0
      random_movement_mode = false
   else
      step_counter = step_counter + 1
      if step_counter > NO_LIGHT_STEPS_THRESHOLD then
         random_movement_mode = true
      end
   end
end
```

The for loop iterate over the robot's light sensors, reading the value of each returned vector and its angle. The values read in polar coordinates are converted using trigonometric functions into their respective cartesian coordinates and accumulated in the respective variables cAccumulatorX and cAccumulatorY.

The vec2_length method is used to calculate the length of the resulting light vector through cartesian coordinates obtained in the two accumulation variables after the iterative cycle. The value of this vector is compared with a threshold to determine if the detection can be considered relevant or not. If no light source is detected for a certain number of steps, upon reaching a predefined threshold, a flag is set that allows the controller to understand that it should enter random movement mode, this is set with a simple boolean flag. The overall detected brightness values that have been inserted into the light_vector vector will later be used in the motor_schemas method, where the vector will be combined with other obtained results.

Obstacle avoidance

```
function avoid_obstacle()
   local proximity_readings = robot.proximity
   local cAccumulatorX = 0
   local cAccumulatorY = 0

   for i = 1, 6 do
      local sensor_value = proximity_readings[i].value
      local sensor_angle = proximity_readings[i].angle
      cAccumulatorX = cAccumulatorX + sensor_value * math.cos(sensor_angle)
      cAccumulatorY = cAccumulatorY + sensor_value * math.sin(sensor_angle)
   end
      for i = 19, 24 do
      local sensor_value = proximity_readings[i].value
      local sensor_angle = proximity_readings[i].angle
      cAccumulatorX = cAccumulatorX + sensor_value * math.cos(sensor_angle)
      cAccumulatorY = cAccumulatorY + sensor_value * math.sin(sensor_angle)
   end

   proximity_vector.x = -cAccumulatorX
   proximity_vector.y = -cAccumulatorY
end
```

In this operation, the values from the robot's front proximity sensors are read and converted into cartesian coordinates and accumulated into variables. Then, the sign is inverted to create a repulsive effect relative to the detected obstacle. A vector is created, which will be used later during a combination operation with other vectors to determine the robot's final movement.

Random movement

```
function random_movement()
   if random_movement_step_counter == 0 then
      local random_angle = math.random() * 2 * math.pi - math.pi
      local random_magnitude = math.random(1, 10)
      random_vector = vector.vec2_new_polar(random_magnitude, random_angle)
   end

   random_movement_step_counter = random_movement_step_counter + 1
   if random_movement_step_counter > 20 then
      random_movement_step_counter = 0
   end
end
```

This function generates a random movement vector. If the random_movement_step_counter is zero, a new random vector with a random angle and magnitude is generated. After 20 steps, the counter is reset.

Movement along walls

The behavior in this case is the same as described in the detailed design section with the subsumption methodology.

```
function wall_following()
   local vtr = {length = 0, angle = 0}
   local value = -1
   local idx = -1

   for i = 1, #robot.proximity do
      if value < robot.proximity[i].value then
         idx = i
         value = robot.proximity[i].value
      end
   end

   if idx >= 0 then
      local angle = robot.proximity[idx].angle
      if angle <= 0 then
         vtr.angle = angle + math.pi / 2
         vtr.length = value
      else
         vtr.angle = angle - math.pi / 2
         vtr.length = value
      end
   else
      vtr.length = 0
      vtr.angle = 0
   end

   return vtr
end
```

The iterative loop scans the robot's proximity sensors and stores the maximum recorded value and the index of the sensor that detected it. With the vector obtained from this operation, it checks if its angle is less than or equal to 0. If so, a new angle is calculated to rotate the robot 90 degrees to the left, added to the detected angle value. Otherwise, if the vector's angle value is greater than 0, a rotation angle of 90 degrees to the right is calculated, added to the detected angle value. The calculated rotation angle is returned along with the value of the maximum vector found. If no sensor has detected a significant obstacle, the returned vector values will be zero.

Vectors combination

The motor_schemas function is responsible for combining the different behavior vectors of the robot (phototaxis, obstacle avoidance, random movement, and wall-following) into a single resulting vector, using specific weights for each behavior.

```
local resultant_vector = {x = 0, y = 0}
local weight_light = 1
local weight_proximity = 2
local weight_random = 0.5
local weight_wall_following = 1.5
```

In this block of code, a vector resultant_vector is defined and initially set to zero, which will be used to accumulate the contributions of the various behaviors. The weights weight_light, weight_proximity, weight_random, and weight_wall_following are assigned to the behavior vectors for phototaxis, obstacle avoidance, random movement, and wall-following, respectively. These weights determine the relative importance of each behavior in the calculation of the resulting vector.

```
local wall_following_vector = wall_following()
local wall_following_vector_cartesian =
    vector.vec2_new_polar(wall_following_vector.length, wall_following_vector.angle)
```

The execution of the method continues by calculating the wall-following movement vector. The wall_following() method is called to perform this calculation, which returns a vector in polar form. This vector is then converted into cartesian coordinates.

```
resultant_vector = vector.vec2_sum(
   resultant_vector,
   vector.vec2_sum(
      vector.vec2_sum(
         vector.vec2_sum(
            {x = weight_light * light_vector.x, y = weight_light * light_vector.y},
            {x = weight_proximity * proximity_vector.x, y = weight_proximity *
                proximity_vector.y}
         ),
         {x = weight_random * random_vector.x, y = weight_random * random_vector.y}
      ),
      {x = weight_wall_following * wall_following_vector_cartesian.x, y =
          weight_wall_following * wall_following_vector_cartesian.y}
   )
)
```

Subsequently, the weighted vectors are summed. The phototaxis vector is summed to the obstacle avoidance vector, both multiplied by their respective weights. This result is then summed with the random movement vector, also multiplied by its weight. The new result is then summed with the wall-following movement vector, again multiplied by its weight. Finally, the resultant_vector is updated with the resulting value. The function returns the resulting vector, which represents the overall movement direction of the robot, based on the combination of various behaviors and their weights.

Actuators control

The actuator function controls the robot's actuators based on the resulting vector combined from various behaviors of the robot.

```
if random_movement_mode then
   random_movement()
else
   random_vector = {x = 0, y = 0}
end
```

Initially, it checks if the robot is in random movement mode: if random_movement_mode is true, it calls the random_movement function to update the value of the random_vector. If random_movement_mode is false, the random_vector is reset to x = 0, y = 0.

```
local resultant_vector = motor_schemas()
local resultant_angle = vector.vec2_angle({y = resultant_vector.y, x =
    resultant_vector.x})
local resultant_magnitude = vector.vec2_length(resultant_vector)
```

After that, the motor_schemas() function is invoked to obtain the resulting vector from the combination of various behavior vectors (phototaxis, obstacle avoidance, random movement, wall_following). From the vector returned by motor_schemas(), the angle and magnitude values are extracted. This resulting vector encapsulates the direction and strength of the robot's movement based on the integrated behaviors.

```
if resultant_magnitude > 0 then
   local left_speed = 10 * (1 - resultant_angle / 4)
   local right_speed = 10 * (1 + resultant_angle / 4)
   robot.wheels.set_velocity(left_speed, right_speed)
else
   robot.wheels.set_velocity(10, 10)
end
```

In the final part, the robot sets the wheels speed based on the vector returned by motor_schemas(). If the magnitude of the overall vector is greater than 0, the wheels velocity is calculated. The division of the overall vector's angle by 4 is aimed at reducing the angle's influence on the wheels speed. Dividing the angle by 4 smooths out the angular response, making the robot's movement more gradual.

If the magnitude of the resulting vector is 0 (indicating no significant direction), the robot sets both wheels to a constant speed of 10, moving the robot straight ahead.

Main controller

The function main_controller() is invoked within the step() function, which is executed at each step of the simulation.

```
function main_controller()
   phototaxis()
   avoid_obstacle()
   wall_following()
   actuator()
end
```

The main_controller() function coordinates the execution of the robot's key behaviors sequentially, ensuring that the robot can navigate effectively in the simulated environment. It responds to changes in light, avoids obstacles, and follows wall contours, all through the movement of its wheels.

4 Experimental analysis

4.1 Definition of objectives and formulation of hypotheses

Objectives

- Evaluate the effectiveness of different instances of control software in the task of phototaxis with obstacle avoidance.
- Identify which control software instance provides the best performance in terms of distance from the light after T seconds.

Hypothesis

- H0 (null hypothesis): There are no significant differences in performance among the different instances of control software.

- H1 (alternative hypothesis): There are significant differences in performance among the different instances of control software.

4.2 Setup

- Task: perform a phototaxis behavior with obstacle avoidance.
- Environment: static but unknown, with randomly generated obstacles and the presence of noise.
- Initial position and orientation of the robot: unknown.
- Evaluation function F: evaluates the distance of the robot from the light source at the end of the simulation.

Procedure

- Generate different instances of the environment and robot positions.
- Perform m independent experiments for each instance of the control software.
- Collect the values of the evaluation function F.

4.3 Data Collection and Analysis

Configuration

- Two instances of control software: Subsumption architecture (P1) and Motor Schemas (P2).
- 100 independent replications for each instance.

Raw data

- Subsumption (P1):

```
[0.13377165778789, 0.11340358866551, 0.15212059728076, 0.16125294998049,
0.13305425273107, 0.072405958968933, 0.16501043344788, 0.13584357245059,
0.16401524647988, 0.15395276565517, 0.14316097796328, 1.4683653463517,
0.14629018074009, 0.13838244585749, 0.13569466902583, 0.12637037195892,
0.16878900193847, 0.13613625793263, 0.15334359483784, 0.12097349562272,
0.16218188741537, 0.14024165247701, 0.14659632509064, 0.13979046525627,
0.17022997558318, 0.1966942082435, 0.14011908659838, 0.13662288331334,
2.1186301015788, 0.16704073180082, 0.13585778960354, 0.12857052820364,
0.16392343945273, 0.14649850225682, 0.1474795286015, 0.88632954030814,
0.15261277485686, 0.13035250290925, 0.17258151008299, 0.14785420380501,
0.16008474901471, 0.1283841370805, 0.16354623545799, 0.118165292788,
0.16806460027437, 0.12358125106552, 3.2353599230397, 0.13803225080191,
0.16416756653975, 0.15393929515156, 0.13314561240079, 0.17910040375175,
0.17223220656268, 0.17195933097535, 0.18273624434913, 0.14366739299775,
0.16591504791838, 0.15674409960771, 0.12990373988382, 0.14742963400132,
0.13074709043611, 0.1198632848667, 0.17467803205909, 0.16876235853317,
0.14094452486631, 0.1141345278736, 4.297077339046, 0.11918436172546,
0.18500848380574, 0.17960311903484, 0.18487913385008, 0.13048649751369,
0.1228821427891, 0.12133806772305, 0.15351655981606, 0.15837097490893,
0.15647754415196, 0.11077972285325, 0.12906263129654, 0.16198812305173,
0.12709335296273, 0.1373151046704, 0.14688820812818, 0.093910655977969,
0.16845442808882, 0.13312223804693, 0.13697570337079, 0.1129253256608,
0.14500622324428, 0.17136380951002, 0.22538135646662, 4.2671378401309,
```

```
0.15654644794443, 0.17978290144011, 0.13314265568172, 0.16513364915466,
0.14149171926473, 0.099040835078281, 0.13252697158304, 0.13596651532203]
(100 values)
```

- Motor Schemas (P2):

```
[0.18081075673993, 0.18100542398751, 0.1811406829394, 0.18130950057567,
0.18123384564009, 0.18109774996099, 2.9972227584877, 0.181222578649,
0.18074623790327, 0.18090450731547, 0.18051441863009, 0.1806032018716,
0.18111744031871, 0.1805578962283, 0.18070878739725, 1.9848324461145,
0.18065754205549, 0.18071789396713, 0.18090332403283, 0.18084513035858,
0.1809040924353, 0.1809026798739, 0.18104921023322, 0.18110221586227,
0.18093769997034, 0.18126129801943, 0.18094256914878, 0.180867941379,
0.18095849556343, 0.18083469097009, 0.18088573701354, 0.18088941466559,
0.18105899207214, 0.18055947539724, 0.18080160697812, 0.18089745241434,
4.438743692108, 1.6187575415004, 0.18099569647912, 0.18068140330698,
3.3798329593147, 0.18117665733791, 0.18107273885905, 3.9667978264834,
0.180855636013, 0.18077215025875, 0.18100144723624, 0.18113900115637,
0.18105225215195, 0.18105301208787, 0.18084113876307, 0.18054754868346,
0.18082547168761, 0.18090938937622, 0.18094817720017, 0.18058173533588,
0.1810494133804, 0.18100038766564, 0.18085370217004, 0.18072184522543,
0.18116552740525, 0.18101049436209, 0.18068434932662, 0.18116195764,
0.18089884560858, 0.18091142293377, 0.18093939067112, 0.18102875858722,
0.1810048680509, 0.18068331342176, 0.18118013354545, 0.18063138636791,
0.18088053868902, 0.18103015933915, 0.18049704662461, 0.18068756259376,
0.18091310852352, 0.18105464379563, 0.18113657942235, 0.18094231157179,
0.18036553635044, 0.18110426770665, 0.18057315572528, 0.18097774532596,
0.18101060763574, 0.18101701039379, 0.18090443722537, 0.18092471396773,
0.07416897681532, 0.1812906267665, 3.3577183279108, 0.18062012693367,
0.18084646964719, 0.18061333092651, 0.18083062670448, 0.18075785589446,
0.18116598522595, 0.18090305756265, 0.18099197895466, 0.18113539284249]
(100 values)
```

Statistic analysis

The calculation of medians and quartiles follows:

- Subsumption (P1):
  - Quartiles:
    - 1° Quartile (Q1): 0.1331
    - Median (2° Quartile): 0.1467
    - 3° Quartile (Q3): 0.1662
- Motor Schemas (P2):
  - Quartiles:
    - 1° Quartile (Q1): 0.1808
    - Median (2° Quartile): 0.1809
    - 3° Quartile (Q3): 0.1811

These values represent the distribution of distances from the light after 30 seconds for the experiments on the two instances.

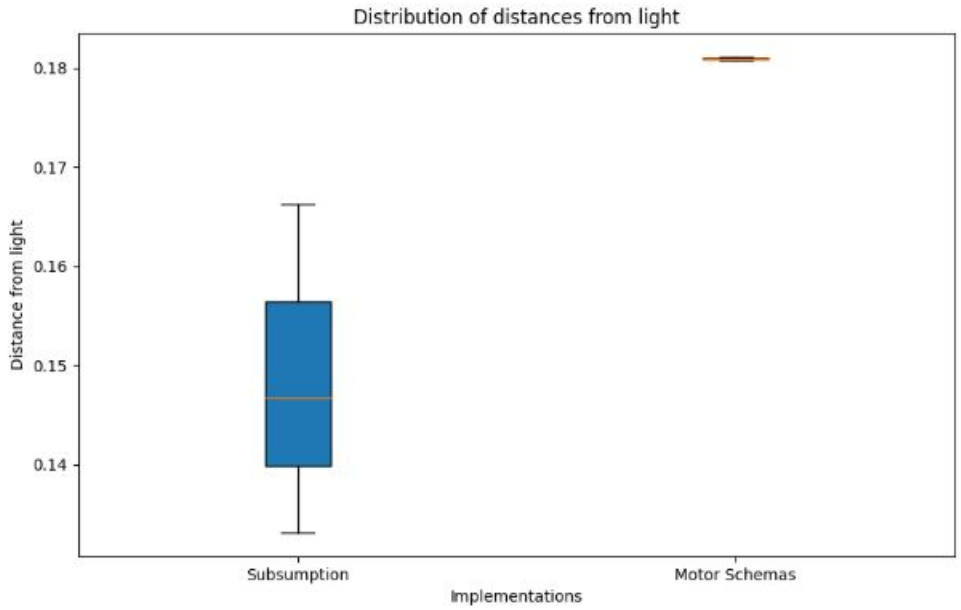

Figure 3: Distribution of distances from light for different implementations

Boxplot creation

The boxplot displays the distributions of distances from the light for the two different implementations, with the respective quartiles and medians indicated.

The graph can be interpreted as follows:

1. Central Lines: the line inside each box represents the median, which is the central value of the data.
   - For Subsumption (P1), the median is 0.1467.
   - For Motor Schemas (P2), the median is 0.1809.
2. Box: the box extends from the first quartile (Q1) to the third quartile (Q3). This range is known as the interquartile range (IQR) and represents the central 50% of the data.
   - For Subsumption (P1), the box ranges from 0.1331 to 0.1662.
   - For Motor Schemas (P2), the box ranges from 0.1808 to 0.1811.
3. Whiskers: the whiskers extend from the edges of the box to the maximum and minimum values that are not considered outliers. For the case where the whiskers are not clearly visible, it indicates that the data is highly concentrated within the IQR.
4. Outliers: any points outside the whiskers are considered outliers. There are no visible outliers in the plot, indicating that there are no extreme values in the data.

Considerations on the results

1. Narrow distribution: the distribution of distances from the light for both implementations is very narrow, particularly for Motor Schemas (P2). This suggests a high level of consistency in the results.
2. Medians and quartiles:
   - The median of Subsumption (P1) is slightly lower than that of Motor Schemas (P2), suggesting that P1 tends to approach the light more closely compared to P2.

- The box of P1 (range between Q1 and Q3) is wider than that of P2, indicating greater variability in the results for P1.

3. Concentration of data: the data for P2 is highly concentrated around the median, with a very narrow box (Q1 and Q3 are nearly equal). This suggests that P2 has very consistent results with little variability.

In terms of performance, both implementations show consistent performance with a very small average distance from the light, indicating that both are effective in the task of phototaxis. The Subsumption implementation exhibits greater variability in results, which could be due to its higher sensitivity to the environment. On the other hand, Motor Schemas shows extremely consistent performance, with a narrow distribution indicating highly predictable outcomes.

Both implementations demonstrate good performance in the task of phototaxis with obstacle avoidance. However, Subsumption exhibits higher variability, whereas Motor Schemas shows exceptional consistency in results.

Evaluation of hypotheses

The Wilcoxon test compares the medians of the data collected from the two implementations. The null hypothesis (H0) of this test is that there are no significant differences between the distributions of the data from the two implementations. The alternative hypothesis (H1), on the other hand, states that there are significant differences.

The Wilcoxon test yielded a p-value of 0.000189. This p-value represents the probability that the observed differences between the medians of the two sets of results are due to chance. The p-value is significantly lower than the common significance level, indicating that the probability of the observed differences between the two implementations being due to chance is highly unlikely. Since the p-value is much lower than the chosen significance level (α=0.05), we can reject the null hypothesis (H0). Therefore, we can conclude that there is a significant difference in performance between the two implementations of control software.

Based on these analyses, it can be concluded that the null hypothesis is rejected in favor of the alternative hypothesis, which suggests the existence of significant differences in the performances of the two implementations.

5 Conclusions

The development of this project has allowed for acquiring new skills in the field of behavior-based controller development, particularly in simulated environments. The knowledge gained could prove very useful in the future, especially if one decides to specialize in the area of behavior-based AI systems.

Based on the conducted tests, the Subsumption architecture implementation has demonstrated significantly superior performance compared to the Motor Schemas implementation in the task of phototaxis with obstacle avoidance. This result could depend on several factors, including potential implementation errors that require thorough review. Alternatively, better parameter tuning could have been achieved using advanced techniques such as F-race or CMA-ES to optimize system performance.

However, it is important to note that this type of implementation leaves room for potential issues. It is difficult to anticipate all possible situations that may occur, and there are specific cases where robots can get stuck in narrow spaces between two obstacles. Occasionally, albeit rarely, they may encounter difficulties in extricating themselves from these situations.

Adopting more sophisticated techniques, such as Behavior Trees or deliberative control methods, could certainly improve the handling of these particular situations. These methodologies allow for more robust and flexible control, facilitating the management of increasing complexity and ensuring more consistent and predictable behavior in unexpected scenarios.