Code Documentation

Swarm Behavior

Introduction

This part of the code documentation describes a solution to the following task: A group of finkens is randomly positioned in a room. The entire space is surrounded by walls. Each finken has sensors pointing in different directions (front, back, left, right, bottom). A sensor measures the distance to a detected object (other finken, wall, floor). Using the collected information all finkens are moving and a swarm behavior should emerge.

Solution

The solution is implemented using V-REP. A basic building block of this framework is the scene. It can encapsulate all models and scripts in a single binary file. This representation of the data is not suitable for source control systems such as git. Consequently, most of the scripts are moved to external files. All external scripts should be added to the V-REP installation folder. An external script (filename script_name.lua) is imported in a V-REP script with the `require "script_name"` statement.

In the concrete solution following files are presented:

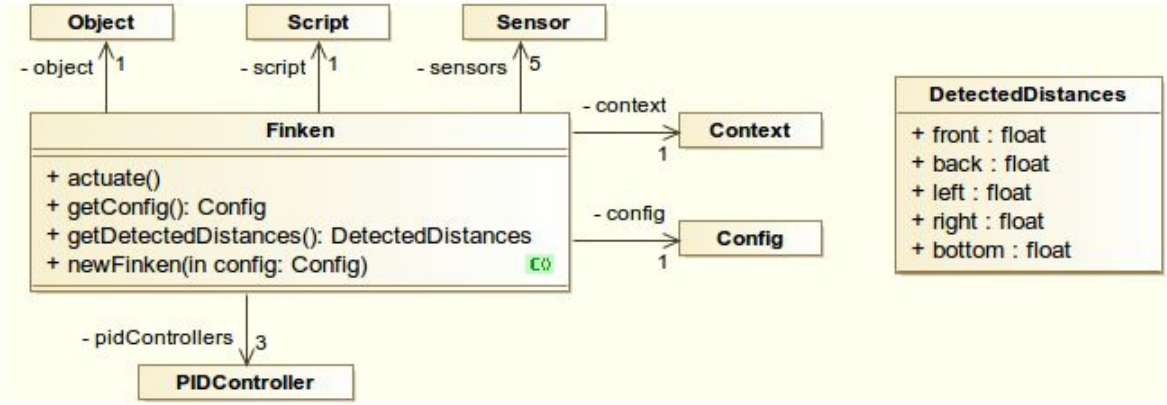
- finken_scene.ttt – a main scene with all models
- finken_utils.lua – a collection of functions describing a behavior of a finken

finken_scene.ttt

There are differences between the finken_scene.ttt scene and the original FinkenTestControlAPIvariables.ttt scene. A ControlScript is created to control the creation and behavior of all finkens. The model of a finken is changed. A bottom sensor is added and the position of the other four sensors is adjusted. Such that, each of them has relative coordinates (0, 0, 0) to the main finken object. This adjustment is important, because the accuracy and efficiency of the calculations is dependent on the position of the sensors.

finken_utils.lua

This Lua script contains an implementation of all functions related to the behavior of a finken. Some V-REP specific functions are used. Lua supports paradigms to encapsulate logically connected parts in a single object. The benefits of this approach originate from the object oriented programming and they improve the development process.



The control over a finken is obtained by the creation of a finken instance e.g.

```
-- For finken with name 'SimFinken#1'
local finken = newFinken{suffix = '1'}
local config = finken.getConfig()
local detectedDistances = finken.getDetectedDistances()
…
finken.actuate()
```

The function `newFinken(config)` can be viewed as a constructor. The `config` parameter is used for configuration and it should contain a valid finken `suffix` as an attribute. All other configuration attributes are optional and they have default values. The configuration can be reached via the `finken.getConfig()` function. All last detected

distances from the sensors are accessed via the `finken.getDetectedDistances()` function. On each simulation step `finken.actuate()` should be called. It executes the main logic: finken control parameters are changed according to the sensors data.

Implementation Details

Finken Object Construction

Each finken instance has following attributes: `config`, `object`, `script`, `sensors`, `pidControllers`, `context`. They hold values, that are used or changed through all simulation steps.

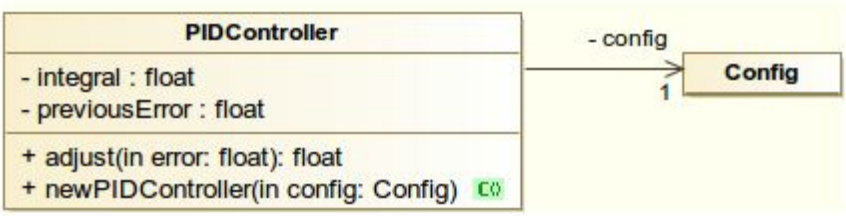
First the `config` object is manipulated with the `prepareConfig(config)` function. The configuration is organized in form of key-value pairs. The function ensures the presence of a value for each key, e.g:

```
config = config or {}
config.height = config.height or {}
config.height.value = config.height.value or 2.4
```

The `config` object contains nested objects. They can have nested objects as well. Hence a hierarchical structure is built, that is traversed with the dot notation. The existence of each object is checked. If it is not initialized, then new object is created. If there is no value for given key, then a default value is assigned.

In V-REP each finken is represented as complex object. Therefor references to specific objects are stored, that can be manipulated later. The attributes `object` and `script` reference the main simulation object of a finken and its associated script. The attribute `sensors` is a collection of references to all sensors of a finken. The function `getObjectHandle(name, suffix)` is used to obtain a reference. The combination between `name` and `suffix` is used as unique identifier for an object.

The attribute `pidControllers` is collection of PID controllers. Each one manipulates different finken control parameter.



The constructor `newPIDController(config)` has `config` parameter, which contains the p, i and d values. They are accessed via `config.p`, `config.i` and `config.d`. The function `adjust(error)` transforms a given `error` into change in the value of a finken control parameter. It uses the attributes `integral` and `previousError` for this purpose. This function should be called on each simulation step. If `previousError` has no valid value (value equal to 0), then the d part is removed from the calculation to prevent unexpected behavior on the first simulation step. If `error` is `nil`, then the controller is reset. PID controllers are tuned manually. In this concrete implementation, only p and d values are used. The tuning is done iteratively by adjusting first the p value and then refining the behavior with the d value. Ziegler–Nichols method was also tried for tuning, but no sufficient results are found.

A finken instance has an attribute `context`. The `context.counters` object contains counters, that are used for different purposes. `global` counts the simulation steps. `wallDetectedFront`, `wallDetectedBack`, `wallDetectedLeft` and `wallDetectedRight` are used for the wall detection. They are implemented as references to objects. Hence each one can be passed as argument to a function, that changes the counter's `value`. The `context.detectedDistances` object contains all last detected distances from the sensors. `context.randomPosition` stores information about the last random position, that the finken is trying to reach.

Finken Object Behavior

A finken (with control parameters pitch, roll, throttle and yaw) is moving in relative coordinate system described by the triple (x, y, z). The function `finken.actuate()` defines the reaction to the changing sensor values.

The function `getObjectOrientation(object)` returns information about the current values for the pitch, roll and yaw. They should be viewed as sensor data.

The function `getErrors(orientation, sensors, context, config)` is using the orientation information and based on the sensors (front, back, left, right and bottom) calculates new relative target for the finken. This change in the position is described by:

- error in x, that changes the pitch
- error in y, that changes the roll
- error in z, that changes the throttle
- error in yaw

The first three cases are controlled via PID controllers with the function `adjustParameter(parameter, error, pidController, config, script)`. `parameter` is the name of the finken control parameter. `config` contains its minimum, maximum and default value. In the main configuration they are defined as:

```
config.roll = config.roll or {}
config.roll.min = config.roll.min or -10
config.roll.max = config.roll.max or 10
config.roll.default = config.roll.default or 0
```

They can be accessed in the function `adjustParameter(…)` via `config.min`, `config.max` and `config.default`. The new value for a finken control parameter is calculated by adding the result from the `pidController.adjust(error)` function to the default value. Then it is compared with the minimum and maximum value and changed accordingly.

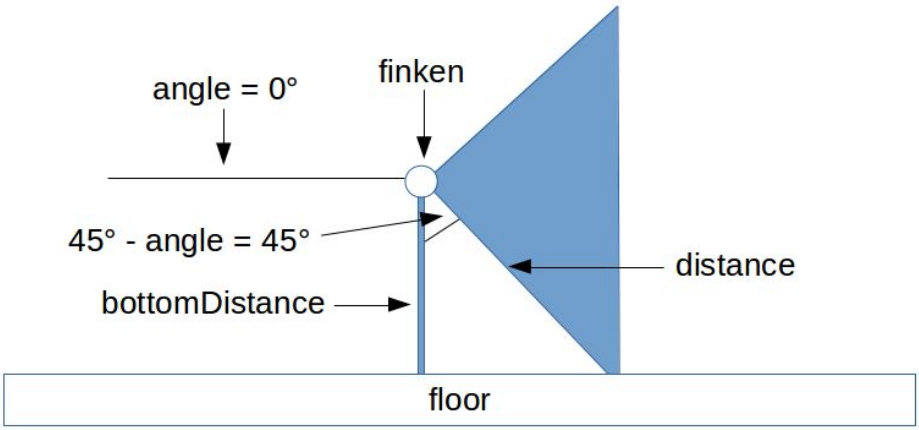
An error in the yaw happens, when a finken is near a wall. The goal is to bring the finken in a stable state by minimizing the amount of sensors, that are pointing to the wall. `adjustYaw(error, orientation, script)` controls the change to the yaw by keeping its value between 0 and 90 degree.

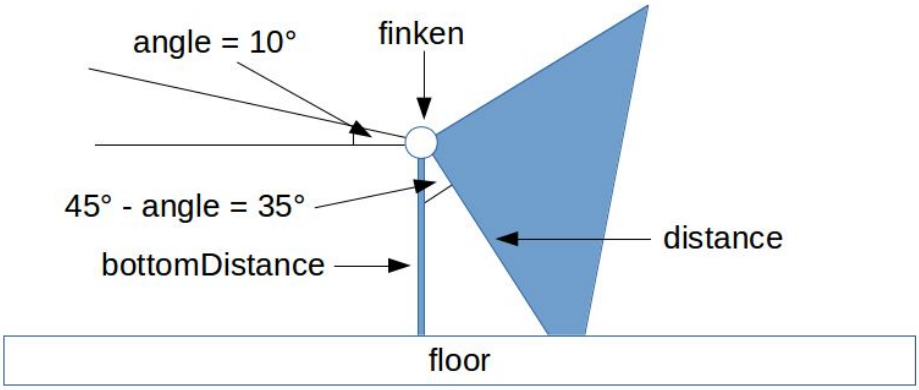
The getErrors(...) Function

The full signature of this function is `getErrors(orientation, sensors, context, config)`. The parameters are described in the previous parts of the documentation. The main goal is to return collection of `errors`, that a finken should try to compensate by changing its control parameters.

First, all sensor data is collected. On the first simulation step, it is possible that some of the sensors are not working. In case of inactive bottom sensor, empty list of `errors` is returned, because most of the calculations depend on the `bottomDistance` value.

A sensor can have a long range. If a finken is flying on small height, then it is possible that the floor is detected by the sensor. The function `isDistanceToFloor(distance, bottomDistance, angle, config)` tries to identify this situation. `distance` is the analyzed value. `bottomDistance` is a distance to the floor from the bottom sensor. `angle` is based on the pitch or the roll of a finken. This depends on the examined sensor. `config` contains configuration values, that control the floor detection process.





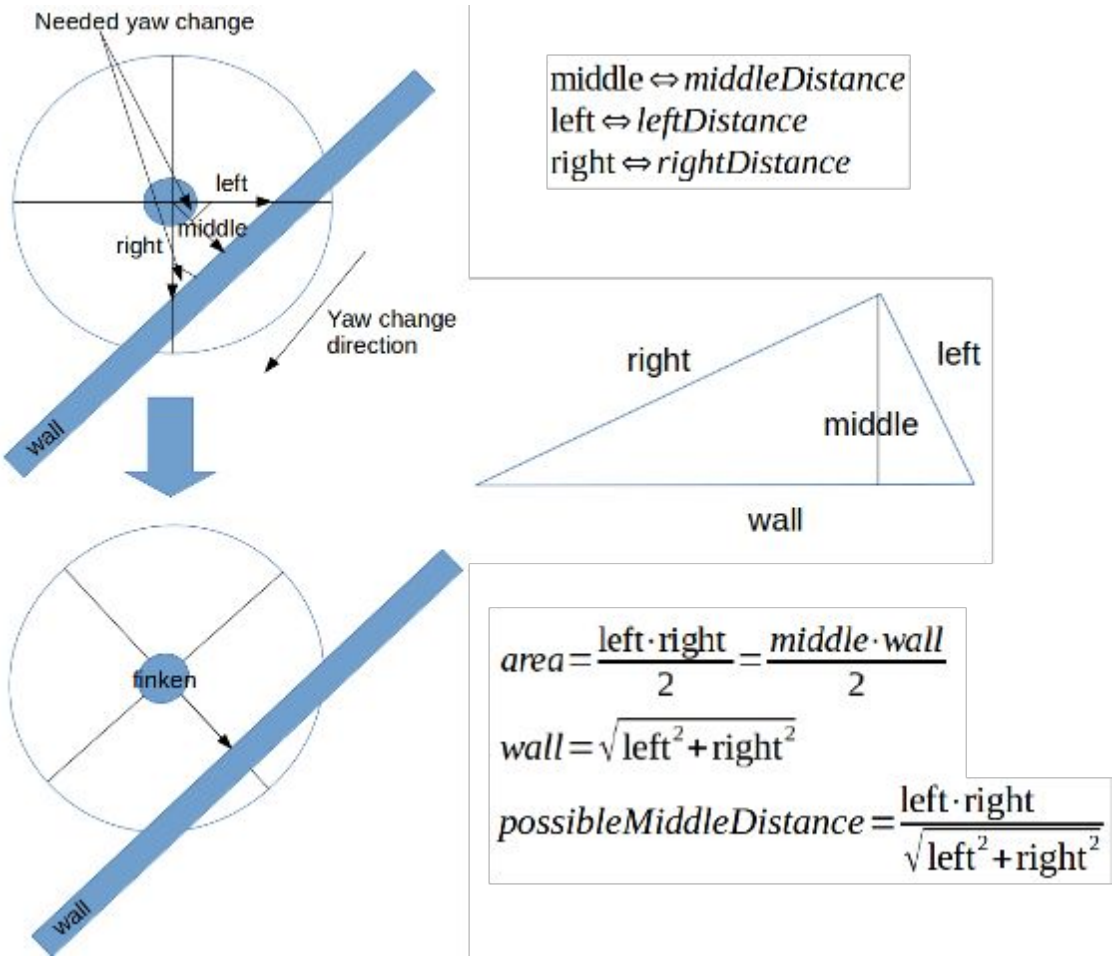
If the sensor is detecting the floor, then `bottomDistance`, `distance` and the floor are building right angled triangle. Based on the Cosine function a `possibleDistance` is calculated and compared with `distance`. If the difference between them is in a given range, then the floor is detected. The described approach expects, that `bottomDistance` is perpendicular to the floor. When the pitch or the roll are changed the bottom sensor is not pointing perpendicularly to the floor. Theoretically a real bottom

distance to the floor can be recalculated based on an approach, that uses Tangent function, pitch and roll angles:

$$realBottomDistance = \frac{bottomDistance}{\sqrt{1+\tan(pitch)^2+\tan(roll)^2}}$$

A solution was tested, but the behavior is not stable, because sometimes the recalculated value is too big (without any logical reason). Therefor a simplification is done. Distance to floor is not recalculated and the range in which a `possibleDistance` is tested has bigger value. In the future the described problem should be investigated, because a stable solution will improve the accuracy with which a `distance` is categorized as distance to floor (may be a low pass filter should be used).

The simulation is executed in virtual room with walls. All finkens are moving. Therefore it is possible, that a finken is near a wall and its sensors are detecting it. All finkens are trying to keep a distance to all detected objects around them. If three of the sensors are pointing at a wall, then the finken will be unstable.



The function `areDistancesToWall(middleDistance, leftDistance, rightDistance, counter, config)` detects this situation. `middleDistance` is the distance from the analyzed sensor. `leftDistance` and `rightDistance` are distances from the neighbor sensors. `leftDistance`, `rightDistance` and the wall are building right angled triangle. `middleDistance` is perpendicular in this triangle. Based on the formulas for area of a triangle and the Pythagoras theorem a `possibleMiddleDistance` is calculated and compared with `middleDistance`. If the difference between them is in a given range, then a wall is detected. `counter` counts the amount of sequential simulation steps when a wall is detected for the analyzed sensor. If it reaches a given value, then yaw is changed. `config` contains configuration values, that control the wall detection process. The needed yaw change angle is determined with the `getYawError(leftDistance, rightDistance)` function, that uses the Tangent function e.g. between `leftDistance` and `rightDistance`. This parameters have the same semantic as in `areDistancesToWall(...)`. After yaw is changed, only two sensors are pointing at the wall and their detected distances have similar values.

The swarm behavior is achieved with the usage of an attraction repulsion function, that is implemented in `getAttractionRepulsionPosition(otherObjectPositions, targetPosition, config)`:

$$\overrightarrow{x_{next}} = w_{target} \cdot \overrightarrow{x_{target}} \cdot a + w_{cohesion} \cdot \sum_{i=0}^{n} \vec{x}_i \cdot \left(a - b \cdot e^{-\frac{|\vec{x}_i|^2}{c}} \right)$$

For given collection of n positions (`otherObjectPositions`) and an optional target position (`targetPosition`, which is generated randomly) it returns next position for the finken. All constants are defined in the main configuration:

```
config.attraction_repulsion.a = 1
config.attraction_repulsion.b = 4
config.attraction_repulsion.c = 1.5
config.attraction_repulsion.wCohesion = 0.8
config.attraction_repulsion.wTarget = 0.2
```

The tuning of this values is done with the following steps: First `c` is set to the required distance between two finkens. Then `a` is set to 1 and `b`, which should be bigger then `a`, is adjusted. Following formula should hold:

$$distance = \sqrt{c \cdot \ln \frac{b}{a}}$$

This values should not be used to manipulate the speed with which a finken is moving. The previously described PID controllers are more suitable for this purpose. The summation of `wCohesion` and `wTarget` should be equal to 1. The second value determines the importance of the random position, that is used to stimulate the swarm movement.

If a finken detects an object, then this object is added to `otherObjectPositions` collection if following constraints hold:

- object is not the floor
- object is not a wall detected by neighbor sensor

A finken is working with relative coordinates. Hence the detected distance can be used as positive or negative, x or y coordinate. This depends on the direction in which a sensor is pointing. The error in z coordinate is determined by the difference between the current and expected height. If a finken is changing its yaw or it is loosing height, then it is unstable. If it is stable, then a random position is generated from all directions, for which no objects were detected. If nothing is detected from the sensors, then the random position is multiplied by given factor. As result, a faster random walk is observed.

The function AddNewFinkenToScene is adding dynamically a new quadcopter to the scene. It will be called, if the SPACE-Key is pressed. The Scene will crash with a stack-overflow error if to many quadcopters are added to the scene.

SSE-Error

The SSE-Error can be used for evaluating different Attraction/Repulsion - configurations. It measures the sum of squared errors between actual distance and desired distance for all proximity sensors.

$$SSE = \sum_{0}^{n} (desiredDistance_n - actualDistance_n)^2$$

The desired distance can be calculated by with this equation:

$$distance = \sqrt{c \cdot \ln \frac{b}{a}}$$

Read the chapter about attraction/repulsion for more informations about the constants a, b and c. The SSE-Errot is plotted to a graph in the scene. If the swarm starts to converge, the SSE-Error gets smaller.

Landscape Searching

Introduction

Another task is to make the Finken quadcopters follow the gradient of a landscape. In the simulation, the landscape is visualized by a texture on the floor. Here, the gradient of a landscape means that the quadcopter follows the lightness of the color and thus, simply a grayscale picture is applied. It is required to have a landscape where the gradient in the neighborhood of the minimum is towards this minimum, like a bowl. Thus, the quadcopter is able to stick to that minimum. Otherwise, the quadcopter could leave the goal area and is not expected to return.

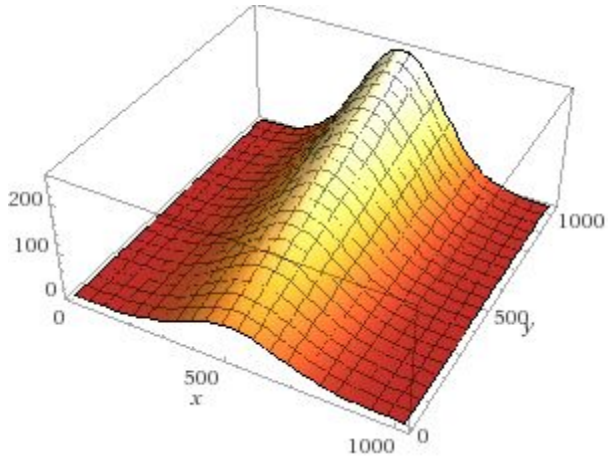
Image Creation

Mathematical formulas and a JAVA program to translate the respective function are used to create a grayscale bitmap. With this approach, any formula could be used and for each differentiable function the gradient is known, at least in which direction the quadcopter should fly. The JAVA-program itself is not complicated, first a new buffered image has to be created, then the color values for each pixel are set and afterwards the image has to be written to a file. Thus, the main part is to design a mathematical formula and fill the bit array with RGB-values between 0 and 255.

One formula has been designed that is a Gaussian curve on the x-axis and a linear function on the y-axis from 25% to 100% lightness. As the image has a resolution of 1024x1024, the formula reads as follows:

$$L(x,y) = a\, e^{-\frac{(x-b)^2}{2c^2}} = \left(\frac{191}{1024}y + 64\right) e^{-\frac{(x-512)^2}{40000}}$$

The peak is given by “a”, which is a linear function in y. The mean is given by “b”, and the standard deviation corresponds to “c”.



3D image of the landscape function

Integration in V-REP

For the integration in V-REP, a new plane has been created to replace the default tiled floor. The image is applied to the floor with a scaling factor of 1024, same as the image size. Then the image has to be scaled along U and V to match the actual floor size, e.g. 5. The ambient color of the floor has to be changed to white to show the real color of the texture.

Attaching the Camera

First of all, the quadcopter always moves randomly. This is achieved by a random walk algorithm. The defined goal color is white i.e., the quadcopter tries to move from black to white or from the dark to the light color, respectively. For consecutive simulation steps, the color gradient is calculated. If the gradient is negative, i.e. dark to light, the quadcopter sticks to its current direction, otherwise a random direction is assigned to the quadcopter. Therefore, attraction is randomly applied to one of the four directions front, back, left or right.

The color sensing in all approaches is done by a vision sensor pointing to the floor. It measures the lightness with a sensor resolution of 1px. Thus, black results in 1 and 0 in white. Thus, white represents the minimum to be reached.

A color sensor on top of the quadcopter is simulated. Hence, an additional shape is added to the top of the quadcopter which is only used as a reference point. The vision sensor is moved according to this reference point on top of the quadcopter and is not rotated. Since the sensor shall not rotate like the Finken quadcopter does, it has to be located outside of

the Finken object hierarchy. The performance of the quadcopter depends on a random walk and thus, the quadcopter not always finds its way to the white color.

Implementation Details

For this task a V-REP scene has been used which is based on a slightly different Finken model. Thus, the Finken actuation is done by calling `move()` instead of `actuate()`, some functionality is grouped in different functions, but the behavior in general is similar.

In order to let the simulation run, the Lua file finken_att_rep_floor.lua is required by the scene landscape.ttt. This Lua file has to be put into a V-REP directory where it can be found, e.g. V-REP3\V-REP_PRO_EDU\lua. Alternatively, the path could be inserted into the scene's control script at the requires statement.

The control script of the scene sets the position of the camera to the position of the dummy sensor on top of the Finken, the sensor itself stays near the floor:

```
local pos = simGetObjectPosition(cam_anchor, -1)
pos[3] = 0.03
simSetObjectPosition(floor_camera, -1, pos)
```

and calls the Finken's actuation function:

```
finken1.move()
```

The file finken_att_rep_floor.lua is responsible for any behavior that is related to the Finken itself. There are instances of PID controllers to control the Finken by roll, pitch and throttle (refer to the function `newPIDController(p, i, d)`) and some functions to do simple calculations e.g., to calculate the scalar product. But the task-crucial part is written in the function `getTargetPosition(object, suffix)`. There, additionally to the horizontal sensors, the vision sensor pointing to the floor is read which provides the information about the actual color value. Analogously to the proximity sensors' detection algorithm, the color gradient towards white can be detected by:

```
if(oldDirection ~= nil and (gradient - oldGradient) < 0) then
      otherObjectpositions = oldDirection
      hasDetectedSomething = true
end
```

If `oldDirection` is `nil` which happens in the first iteration, then there is no `oldGradient`. Therefore, this block is not executed. The gradient towards white is given by:

```
(gradient - oldGradient) < 0
```

Thus, in this case attraction and repulsion can be applied to make the Finken follow the gradient as described later. If the quadcopter does not fly towards the white color (or it actually is the first iteration), then the random walk algorithm is applied. That is if no detection is made, the Finken gets attracted towards a random direction (front, back, left, right). Additionally, this is restricted to each 5th iteration to make the Finken fly stably.

Finally, for object or random attraction the target position of the Finken is calculated. If several objects (other Finkens or walls) are detected, attraction and repulsion are determined for each Finken-object distance by the sum over all attraction repulsion values. `oldGradient` and `oldDirection` store the last color and the list of the attracted direction(s) for further use in the next iteration.