Heidelberg University
Automation Laboratory

Final Report for Robotic Games

CATching the Mouse

Maximilian Klingmann, Julien Stern, Zhaboin Zhu, Chenyang Zhu

supervised by
Holger Dietrich

Matriculation Number: 2932908, 3057886, 3399494, 3573590
Date of Submission: 14/08/2019

We, **Maximilian Klingmann, Julien Stern, Zhaboin Zhu, Chenyang Zhu**, hereby declare that this report, titled **CATching the Mouse**, written for the **Automation Laboratory** and supervised by **Holger Dietrich**, and all material presented were created by ourselves entirely. Furthermore we declare that all adaptions from other sources are specifically acknowledged throughout the thesis. Citation, as well as the usage of foreign sources, texts or any other aid are denoted according strict scientific rules. We understand that we must not present foreign texts or text passages. Doing so is seen as cheating and violates basic rules of scientific working. We are aware that cheating would lead to a withdrawal of the examination among other consequences. We declare that all statements and information contained herein are true, correct and accurate to the best of our knowledge and belief.

Heidelberg, 18/9/2019

Contents

2 Theoretical Background

(Maximilian Klingmann)

In the following chapter we elaborate on the theoretical background that is needed to understand the functionality of our robot. First we work out the goal of the game that the robot wants to participate in. We then go into detail about the *Recursive Nested Behavioral Control* and how our robot handles the fusion of different behaviors to conclude the most promising strategy to win before diving into the technical details of the robot and how commands can be issued to it in the next chapter.

2.1 The Game - Catching the Mouse

In the game our robot participates in, there are two players, one being our robot and one being another robot of the same size and model. Two roles, i.e. the *Mouse* and the *Cat* were assigned to the robots respectively. The goal of the cat was to catch the mouse before it could reach a pre-defined destination denoted as the *Cheese*. There were several hiding spots the mouse could use as cover to hide from the cat which constantly tried to find the mouse. A match was over when either the mouse caught the cheese or the cat caught the mouse. Each group was assigned either the implementation of the cat or the mouse robot. We decided on the cat and thus the main goal for our robot was to catch the mouse as fast as possible.

2.2 Recursive Nested Behavioral Control

In the past there have been two models that were widely used in mobile robots. One being the so called *vertical decomposition*, also known as *subsumption architecture*, first introduced by Brooks [1] and others. Although it already followed a bottom-up paradigm it only relied on the sensory data received

to react to. The other being the *horizontal decomposition* introduced by Crowley [2] and others. However this architecture relies on an a priori known model, e.g. a map of the surroundings to pre-plan paths and compile a behavior from that data. In modern days a combination of the both called the *Recursive Nested Behavioral Control (RNBC)* is used. It also follows the bottom-up paradigm of the former of the two previous architectures but it can also benefit from the horizontal decomposition in higher levels. In other words it is believed that both architectures are needed to realize an autonomous mobile robot.

In the RNBC each layer of the bottom-up design denotes a level of abstraction from the robot with the first layer handling the input of raw kinematics and dynamics of the robot. Further up are more general tasks such as collision avoidance, homing or even path planning, which is where the latter of the two previous architectures might come into play. See Figure 2.1 taken from the lecture notes for a visualization of the RNBC. One can clearly see the bottom-up design with the more abstract tasks being further up. Furthermore the attentive viewer can see that there are only connections between neighboring layers, thus the recursive nature of the architecture. These links are called forward- and back-feed respectively. Each layer can also have feeds from sensors, such as a camera a gyroscope or like in our case a sonar sensor.

2.3 Behavior Fusion

Because each layer of the RNBC suggests a specific behaviour the robot should follow next a *Behavior Fusion* is needed, that is in specific stages the robot needs to react differently. However the decision which behaviour is the best eventually is not trivial and needs to be thoroughly discussed. If, for example, the cat has a layer that is connected to a camera input and that layer can detect the mouse it might suggest driving directly towards the mouse, as the ultimate goal for the cat is to catch the mouse. However, it might not detect if an obstacle is between it and its prey and thus the robot would drive into that obstacle. To avoid this behaviour another layer, i.e. the

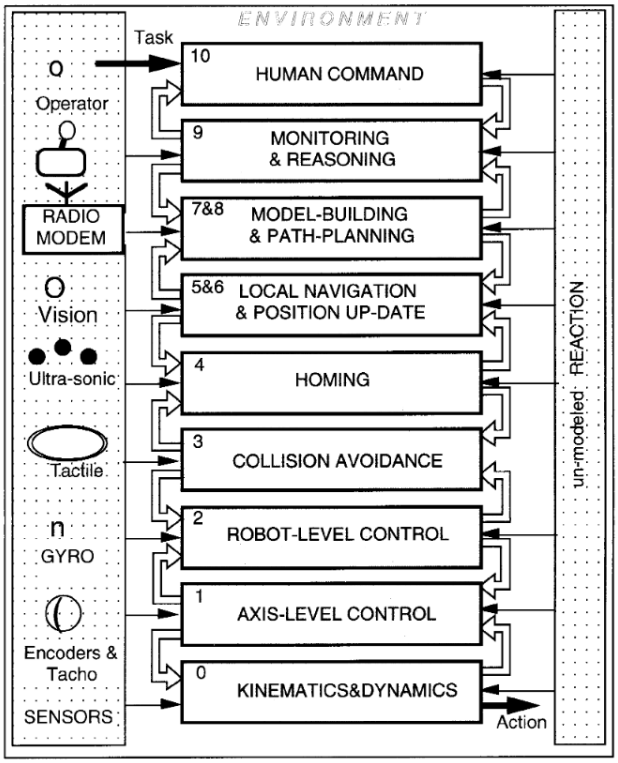


Figure 2.1: The bottom-up design of the RNBC

collision avoidance layer which reads data from a sonar sensor to measure distances to close objects suggests another behaviour, that is driving around the obstacle before catching the mouse. The two behaviours received need to be fused together in a gate before passing the ultimate behaviour. In general there are different kind of gates used for that fusion with one example being a default OR gate where each input behaviour is weighted according to their magnitude to result in a wanted fused behaviour. In the absence of one input the output tracks the other input.

Another example is the prevail gate where one input is weighted exceptionally higher than the other. The lower prioritized one is passed through, if the first input is absent. However when the higher prioritized one is present it is strongly dominant to the other. This is the gate we used several times in our behaviour fusion. Figure 2.2 illustrates the usage of the prevail gate in our cats behaviour fusion. It is shown that the behaviour suggested by the

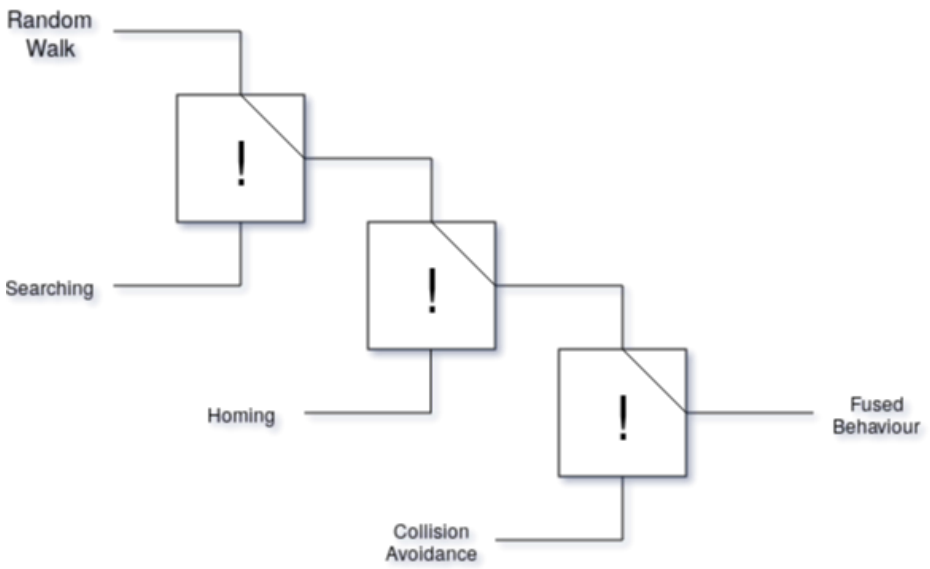


Figure 2.2: The behaviour fusion model of our cat robot. The first prevail gate shows that searching is more important than random walk when we recently lost the mouse. The second prevail illustrates that when seeing the mouse the homing layer is more relevant than any prior layers. The last prevail shows that even if we see the mouse we need to avoid obstacles, thus collision avoidance is more prevalent.

random walk layer has minimum priority as it is connected to the behaviour of searching at the very first level. The fused behaviour from these two inputs is then tested against the behaviour homing suggests in yet another prevail gate. In this case the suggested behaviour coming from homing has higher priority, than the fused behaviour coming from random walk and searching. The resulting fused behaviour of homing and the formerly fused behaviour of random walk and searching then again serve as an input for another prevail gate where the counter part input is the behaviour suggested from collision avoidance. This results in a hierarchy of importance, where random walk is the fallback behaviour if nothing else is applicable and collision avoidance being the most important suggested behaviour, thus our robot cat following a mouse would still avoid collision with nearby objects rather than following the mouse in a straight line wherever it goes.

3 Technical Details

After we clarified the theoretical background behind the RNBC and the Behavior Fusion in the previous chapter a brief overview of the used robot and its technical details will be given in the following sections. We introduce the Pioneer 3DX and how it is operated. At the end of the chapter we briefly expound on the way we incorporated the RNBC into our implementation, however detailed explanations of each layer are given in following chapters.

3.1 The Pioneer 3DX

The robot used in this project is the Pioneer 3DX which is operated through the *Robot Operating System (ROS)*, an interface library to send messages to the robot. However the full technical details of ROS are beyond the scope of this report, so we briefly explain the most important terminology used by ROS. The forward- and back-feed connections used in the RNBC are attained with ROS topics, which function as channels where messages can be published and received by the layers. These topics are also used by the sensors to publish their information, such as camera feed, sonar distance data and so on. In our case we created separate scripts which represented the layers of the RNBC.

3.2 The RNBC of our cat

As described in the previous chapter the RNBC follows a bottom-up design beginning with the pure kinematics and dynamics at the very first layer. However in our case a few layers were provided by the lecturer. Our entry point to the robot was a velocity command that we needed to publish. The velocity command consists of a linear and and an angular velocity that we want to pass to the robot.

Because of the reason a couple of layers were provided we decided to start our RNBC model with a *Collision Avoidance* task. In this collision avoidance task we take the sensory data of the sonar scanner to find nearby obstacles and avoid these. Next in our hierarchy was the *Homing* task, which in our case is the stage where we detect the mouse and suggest a behavior where we follow the mouse. For that we use the feed of the USB camera mounted on the robot. When we lose the mouse after seeing it our next layer, the *Searching* layer, is activated. This layer is the next above homing and it suggests a behavior to search for the mouse at locations we recently saw it at. Our last layer is the *Random Walk* as the most abstract and loose behavior. Whenever there is no collision pending, we do not see the mouse and we haven't seen it recently we roam the field by a random walk. The RNBC of our cat robot can be seen in Figure 3.1. The RNBC type design can clearly be seen. The squares denote topics, forward- and back-feeds and feeds from sensors. The squares denote the layers of our RNBC previously described. The topic furthest down in the image shows the velocity topic provided to us by the lecturers.

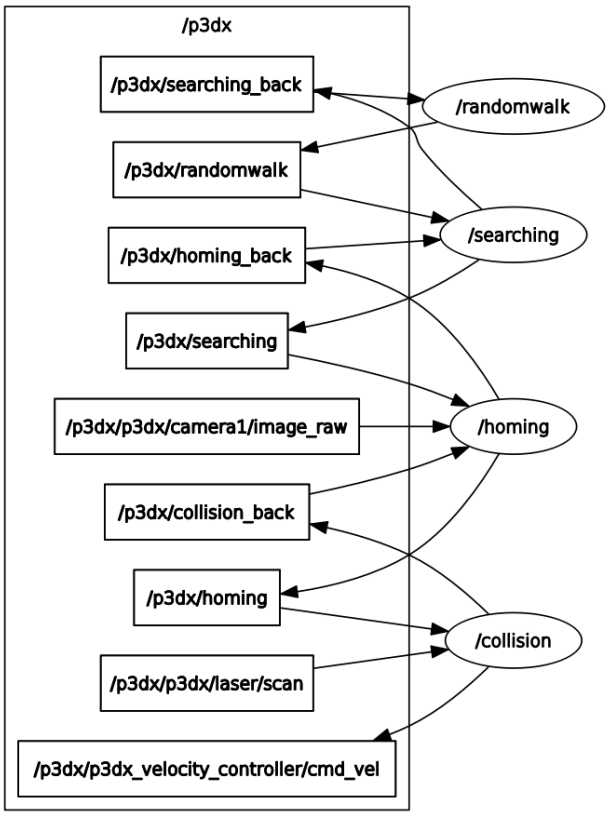


Figure 3.1: The RNBC of our cat robot

4 Implementation Details

After clarifying the theoritcal background and the technical details of the robot as well as the general concept of our robot we now delve into the implementation details of the four layers described before. Since the RNBC is a bottom-up design we also start with the layer furthest down in our RNBC, which is the collision avoidance. We show when a collision gets detected and how it handles the avoidance before going into detail about how we detect the mouse using the data stream provided by the USB camera. Lastly we illustrate what happens when the mouse gets lost and how the robot reacts to the circumstances when it doesn't know anything about the whereabouts of the mouse.

4.1 Collision Avoidance

(Zhaobin Zhu)

4.1.1 Approaches

The development of control mechanisms for autonomous robots has been a widely researched field for years. Some applications require a robot to act independently in a more or less familiar environment. However, there is still no general method for robot navigation and collision avoidance. One of the reasons is both the areas of application and the architectures of the robots vary greatly.

According to Javier Minguez et al. [4], navigation systems can be divided into three different categories: Model based schemes, reactive schemes and hybrid schemes. With model based schemes, motion commands can be extracted directly by a model from the environment. Reactive systems works according to a "perception-action" cycle, where the information of the environment provided by sensors can be interpreted as "perception" and the

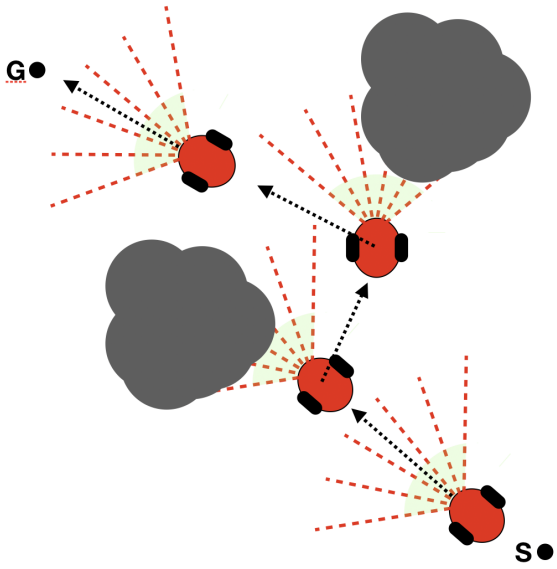


Figure 4.1: Collision Avoidance for P3DX

movement information as "action." Therefore, such systems must be navigated step by step, since they can only perceive their local environment. The hybrid scheme combine both schemes.

However for this project the robot doesn't know global environment and should move autonomously and safely, therefore local collision detection and collision avoidance are fundamental properties, which should be implemented at the beginning of the project. There are already numerous approaches to collision avoidance for reactive systems. In the following, the best-known algorithms are briefly explained.

Bug Algorithm: Is probably the best-known algorithm, because the Bug strategy always finds a way from the starting point S to the target G if one exists. Until reaching the target, the robot first runs towards the target, until an obstacle is detected at point A. Then the robot circles around the obstacle and notes that point B has the smallest distance to the target. After that, the robot goes back to point B with the shortest distance and leaves

the obstacle and drives towards the target.

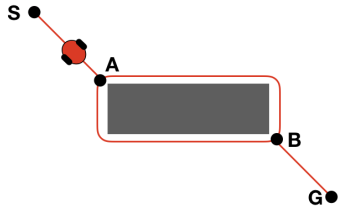


Figure 4.2: Bug-1 algorithm

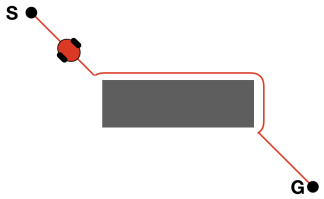


Figure 4.3: Bug-2 algorithm

Since the Bug algorithm is not very efficient, it will be improved in Bug-2 algorithm. Instead of going back to point B, the robot immediately leaves the edge of the obstacle and moves in the target direction. [6]

The Vector Field Histogram (VFH) Algorithm: is another method for obstacle avoidance in real-time applications on mobile robots. This method is based on a two-dimensional histogram grid, which is continuously updated by the current sensor data. The higher entry in the histogram grid corresponds to the presence of an obstacle in the environment or the robot is far away to an obstacle. Contrariwise a lower entry indicates a free area or the robot is closer to an obstacle.

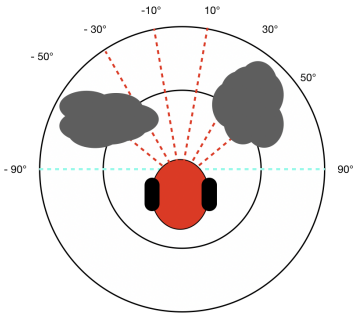


Figure 4.4: Polar representation

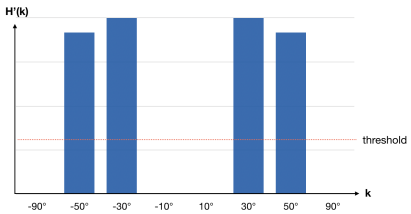


Figure 4.5: Polar histogram

At the same time, the "Valley" in the Polar Histogram represent passable areas. With a defined threshold can now be determined from which threshold

value a mountain in the polar histogram is recognized as an obstacle, or from when a valley is recognized as a free direction. [6]

P3DX - Sonar

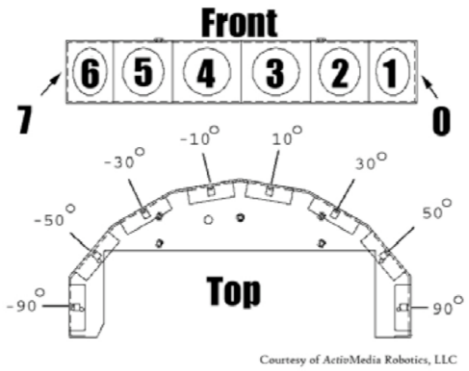


Figure 4.6: positions of ultrasonic sensors [3]

As already described at the beginning, the perception of a reactive system is made possible by sensors. Depends on type of robot and sensors, there are many techniques to detect collisions. Based on the specification of P3DX, we know, that P3DX robot provides up to four ultrasonic arrays with each has eight sensors, which is covering an angle of 180 degrees. Every sensor has a fixed position on the robot, which could be accessed by the index from zero to seven. The index is in order from left to right, which means the 0th index is on the side of the positive aches, and the seventh index is the negative x aches. Except for those sensors on the side, each sensor in the middle has a range of 20 degrees. On default, the measurement could perform at a frequency of 25 Hz, which equals to 40 milliseconds each measure. With the returned data of those sensors, it is possible to make a simple object detection or use the range information to avoid collisions. [3]

Implementation

In the ROS system, the communication between different layers is usually realized by the publisher and the subscriber. Since we decided on the

"Recursive nested behaviors control" control structure, communication only takes place between two adjacent layers. Therefore collision avoidance publishes depending on whether it is a simulation or a real application, on p3dx_velocity_controller/cmd_vel or on /RosAria/cmd_vel. A twist object is sent as a message, which contains velocity information and can be interpreted directly by the cmd_vel. Additionally, collision avoidance publishes an Int64MultiArray object on the topic /collision_back, which informs the higher layers that a collision has been detected.

```
if args.sim:
    pub = rospy.Publisher('/'+args.robot+'/p3dx_velocity_controller/cmd_vel',
    Twist, queue_size = 1)
else:
    pub = rospy.Publisher('/RosAria/cmd_vel', Twist, queue_size=1)

bpub = rospy.Publisher('/' + args.robot + '/collision_back',
Int64MultiArray, queue_size = 1)

def listener():
    rospy.init_node('collision')
    if args.sim:
        rospy.Subscriber('/'+args.robot+'/'+args.robot+'/laser/scan',
        LaserScan, col_callback)
    else:
        rospy.Subscriber('/RosAria/sonar', PointCloud, col_callback)
    rospy.Subscriber('/'+args.robot+'/homing', Twist, hom_callback)
```

On the other hand, the robot either listened to /laser/scan topic or /RosAria/sonar topic. Depending on the topic, the distance information is received either as PointCloud object or as LaserScan object. At the same time, the corresponding collision_callback function is called, which processes the received messages. As already mentioned, collision avoidance must also listen to the higher layer. This is realized by an extra subscriber, which listens to the topic /homing and triggers the homing_callback function.

As can already be seen from the VFH algorithm, the accuracy of collision detection and corresponding collision avoidance ability of the VFH algorithm depends massively on the coverage of the sensor, measurement rate and the number of sensors. Therefore, our team has decided to implement our own algorithm. Because the P3DX robot does not allow motion parallel to the common axis of the drive wheels, we also decided to leave both sensors along the y-axis and only use the sensors from -50 degrees to 50 degrees in our algorithm.

```
COL_DIST = 1.0
```

```
    if args.sim:
        collision_left = [msg.ranges[angle] < COL_DIST for angle in [80, 120,160]]
        collision_right = [msg.ranges[angle] < COL_DIST for angle in [200, 240, 280]]
        distances = [msg.ranges[angle] for angle in [80, 120, 160, 200, 240, 280]]
    else:
         distances = [point_norm(p) for p in [msg.points[1]...msg.points[6]]]
         collision_left = [point_norm(p) < COL_DIST for p in
         [msg.points[1],msg.points[2],msg.points[3]]
         collision_right = [point_norm(p) < COL_DIST for p in
         [msg.points[4],msg.points[5],msg.points[6]]

if any(collision_left) or any(collision_right):
        if not avoidance_active:
            len_col_left = len([i for i in collision_left if i])
            len_col_right = len([i for i in collision_right if i])
            direction = -1 if  len_col_left > len_col_right else 1
        twist = Twist()
        twist.linear.x = min(distances)/COL_DIST;
        twist.linear.x = twist.linear.x if min(distances) > 0.6 else 0
        twist.angular.z = (1 - abs(twist.linear.x)) * direction

        pub.publish(twist)
        avoidance_active = True
else:
    pass_through_homing()
    avoidance_active = False
bpub.publish(Int64MultiArray(data=[-1,-1,-1, -1, 1 if avoidance_active else -1]))
```

First at all we convert the measurement results of the laser-scan data for a simulation or the sonar data for real application into polar coordination. But instead of using the collision probability like in the VFH algorithm, we decided to store the absolute values as distances in the distances array. Our Team also defines an minimum distance on top of the Robot radius, where the robot can move safely and freely within this environment. If the value falls below the predefined threshold, the collision avoidance will be triggered.

However, it is not easy to define the threshold for the minimum distance. If the range is chosen too big, the robot can no longer drive through two obstacles. But on the other hand, if the threshold is chosen too small, the guaranteed collision behavior can no longer be performed. Because there is not enough space to maneuver and the robot may not be able to make sharp turns at high velocities. To find an ideal threshold, we performed two test scenarios with different constellations on the robot. In the first scenario, the robot must pass through a kind of gate, consisting of two geometries without touching the geometry. In the other scene, our robot is placed in a situation where it was surrounded by several geometries and had to free itself from the case.

As we can already see from the Bug algorithm, the Bug-2 algorithm is

more efficient than the first Bug algorithm, because it leaves the obstacle at the shortest distance to targets and drives to the target. Because our target is continuously moving, we cannot assume that the target is still in the same position after collision detection. For this reason, our team has decided as soon there is no collision detected, the target information from homing has to be forwarded to the velocity controller. On the opposite, if a crash would occur, homing is neglected for the time being.

For collision avoidance, the next step is to iterate through the histogram generated by the sensors and calculate the total number of vectors below the defined threshold. Depending on which of the two arrays contains more elements below the threshold, the direction of rotation is determined first. Subsequently, linear velocity in the positive x-direction is determined as a quotient of the smallest value in the histogram and the minimum distances that triggers a collision. Using the quotient can be deduced that the further away the robot is from an obstacle, the more it can maneuver in the x-direction during rotation. As an additional safety feature, our team decided, if the distance to an obstacle is less than half the minimum safety distance, the robot should only rotate around the z-axis and has no linear velocity. To get the effect, when the robot is away from the obstacle, the robot should have more linear velocity than angular velocity. On the opposite, if the obstacle is near, the robot should have more angular velocity than linear velocity. For this ability, the rotation velocity is calculated on the linear velocity in the x-direction. To keep a clear separation between the respective control layers as already mentioned before, a "geometry-msgs.msg" is created for the motion command, which is finally published on the topic of velocity controller.

But there is also the possibility that both arrays have the same number of vectors, which undercuts the threshold for collision avoidance. So the robot would usually remain trapped in one state and can't move to other stats. To counteract this, we have decided that if both collision-left and collision-right with the same number of vectors are triggered, the robot will rotate until the equality of the vectors is no longer given and will then be freed from the situation.

4.2 Homing

(Julien Stern) In this section we describe our Homing procedures, being the identification of the mouse robot, calculations of the relative position of the mouse robot and the resulting angular and linear velocities. Thus, homing is the procedure, where our robot discards velocity commands coming from the Random Walk or Searching strategy, since these have lower priority in the event of a found mouse. Restricted only by the commands of the Collision Avoidance strategy - to avoid collision is always the top priority - the robot tries to follow the mouse as long as the Homing strategy decides that the mouse is in sight.

4.2.1 Identification of the target

The first step in the Homing strategy is processing the raw data that comes from the only visual sensor installed on the robot: The camera feed. This feed is a raw image sent to the robot multiples times per second. The goal of the identification task is to decide whether or not the mouse is present in an image. For this object tracking task many methods can be employed. We tested an Neural Network based object detector, an optical flow approach and a color based method involving masking and morphological transformation procedures. Eventually we decided to implement the latter, since it proved robust and relatively easy to implement.

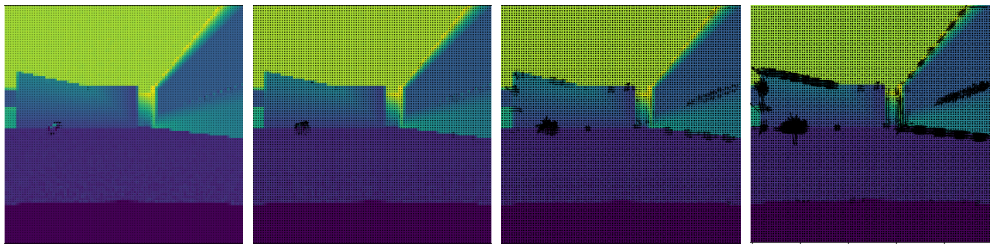
Object Detector Firstly we tested an Object Detector called *YOLO 9000* [5] pre-trained on the *ImageNet* data set which is a large real world image data set with over 9000 labels. Initial testing resulted in the robot being recognized as an object of class *fire engine* but only from certain perspectives. Perspectives not showing the wheels properly lead YOLO to not classify the robot at all. Figure 4.7 shows and example images, where YOLO detects the robot and also its correct position. Since computation time is a limiting factor in real time applications like the mouse and cat game our robot is competing in, we depend on fast image processing, optimally multiples times

Figure 4.7: YOLO detecting the robot as class *fire engine*.

per second. YOLO however, if compiled as CPU version, takes significantly more time to process one image. The reason behind this is its relatively deep structure and the fact that we can not rely on fast GPU compiled versions. Hence we dropped the object detector approach and tested other methods.

Optical Flow Optical Flow based on the *Lucas-Kanada* method is a technique that tries to predict the relative movement of an object frame by frame. It does this by comparing two consecutive images at given grid points and predicts the movement based off of neighboring pixels. This method works well in environments where the camera itself is stationary. Figure 4.8 depicts the problem with optical flow in its basic form: As long as the camera does not move, the largest vectors in the image are the ones caused by the other robot, so a prediction of its position is possible. However as soon as the camera itself moves, edges of walls begin to also have larger flow vectors. These errors can be cancelled out by calculating the viewers own movement and correcting the vectors that way, but we decided against this augmentation since we wanted to keep the Homing identification process as simple as possible to sustain error robustness and short computation times.

Color Based Method The last method we tested and ultimately chose as our target identification process was a color based approach since we knew that the other robot will have red as its primary color. The images received



(a) (b) (c) (d)

Figure 4.8: Optical Flow visualized by black vectors pointing in the direction of the flow. a) other robot starts to drive, depicted by the larger vectors to the left, b) other robot is driving, our robot starts to drive, c) both robots drive, d) both robots drive with their normal speed.

from the camera topic initially are in the RGB format. This format is very light sensitive and not easy to process and to apply a human readable color range on. Hence, the first step in this approach was to convert the image into the *Hue Saturation Value* (HSV) color format. This allowed for more light independent and intuitive color range adjustments during testing. Secondly we applied a mask onto the image which filters out all non-red pixels. By designing this step as a range of color, saturation and brightness we could successfully separate most of the mouses hull from the rest of the image. This masking step is depicted in Figure 4.9a. Consecutive morphological transformations were added to the masked image. First an opening step was applied that removed outliers (see Figure 4.9b). After that a closing step was performed to close small black holes and gaps between or inside the white parts (see Figure 4.9c). Figure 4.9d finally shows the resulting bounding box (red) and its centroid (green) around the contours (blue). The cyan dot marks the center of the image and becomes relevant later on in the process. Since this method can result in multiple bounding boxes we decided to only use the biggest one present to use in further computations.

4.2.2 Localization of the target

Given the predicted bounding box off the moues robot the relative position of it must be calculated in order to move towards it. By measuring the

(a) (b) (c) (d)

Figure 4.9: The four stages of the bounding box localization process.

horizontal distance between the center of the bounding box and the center of the image and normalizing this distance with half the images width we end up with a distance $d \in [-1, 1]$, where negative values indicate that the bounding boxes center is to the left of the images middle line and vise versa for positive values. This distance is used in the next chapter to calculate the resulting velocity in order to reach the target robot.

4.2.3 Calculting the resulting velocity

(Chenyang Zhu) The entire homing consists of object detection and turning control algorithms. In order to simplify the procedure, we initially use the open-loop control method to give the steering angle of our cat robot, according to the distance difference between the direction of the cat robot and the center position of the detected mouse through the object detection, but since the mouse is usually also moving at the same time, it is easy to cause the cat robot to lose the signal of detecting the mouse before it catches up with the mouse. This situation is due to the lack of closed-loop control, the controller does not get feedback after the output control signal, so we improved the controller structure, using a closed-loop controller. After analyzing the requirements of the competition, we chose the PID controller as the kernel output controller. A PID controller is a controller that controls the proportional, integral, and derivative effects of control system deviations. PID control is a control algorithm that integrates information about the “past”, “now” and “future” of system deviations. PID control embodies the fastness, stability and accuracy characteristics required for automatic control. PID controller has a simple structure and is easy to implement. It also has good stickiness and high

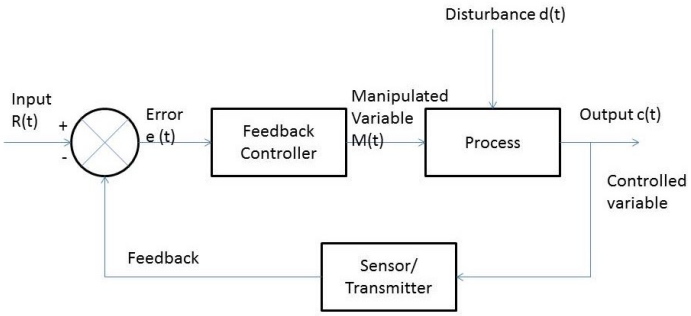


Figure 4.10: The structure of feedback control.

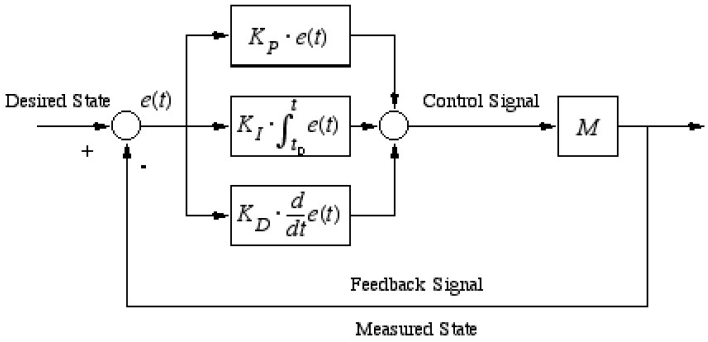


Figure 4.11: The structure of PID control.

reliability.

The conventional PID control system mainly consists of two parts: the controlled object and the controller. The process of PID control is: the control system makes the difference between the input value and the output value at the input end, obtains the system deviation, performs proportional integral and differential operation on the deviation, and gives the results of the three operations to the corresponding gain processing respectively, and then the processed result add the total control amount, and then control the controlled object to form a PID controller. The structure of a common analog PID control system is shown.

There are three parameters in the PID controller, which are proportional coefficient, integral time constant and derivative time. The function of real-time PID is shown below.

$$U(t) = Kp\left[e(t) + \frac{1}{T_i}\left(\int_0^t e(t)\,dt\right) + T_d\frac{e(t)}{dt}\right] \tag{1}$$

And this is the function of PID controller we used in our program.

$$TURN\_DIR = K_p e + K_i \sum_n e + K_d \Delta e \tag{2}$$

Usually these three constants have different effects on the dynamic and steady-state performance of the system. The values of these three parameters are the most important part of PID controller. It determines the control effect of the PID control system.

Scale factor The proportional action linearly reflects the deviation signal of the system. After the deviation occurs, the control effect is generated at the fastest speed to reduce the deviation. The proportional action reflects the rapidity of the PID control. The larger the proportional control parameter Pk, the faster the control system moves, and the more the oscillation times, the longer the adjustment time. When Pk is too large, the system will tend to be unstable. If Pk is too small, the system will move too slowly and the adjustment time will become longer. In the specific robot test, the robot will immediately output a steering value according to the error. When the proportional control parameter is small, the robot will take a long time to track the mouse. When the parameter is too large, the robot reacts very quickly. However, it will continue to turn back and forth, which is a big damage to the robot's own motion system.

Integral time constant The integral action is mainly to eliminate the steady state error. The integral action reflects the accuracy of the PID control. The main performance of the system is as follows: the integral action will affect the stability of the system, smaller the T_i is , less stability the system had, and when the T_i is too small, the system will oscillate; If T_i is too large, the impact on system performance will be reduced. Only when T_i is suitable, will there be ideal characteristics. When debugging the robot's controller, the integral action is mainly to make the robot adjust the steering more

quickly, but the introduction of the integral action will lead to the problem of integral saturation, which may cause the robot to fall into a state in which it cannot be turned in the other direction, so it is necessary to take some Anti-saturation measures.

Differential time constant The differential action reflects the stability of the PID control. According to the trend of the error signal, the dynamic characteristics of the closed-loop system are improved by the advance action. When T_d is too large, there will be a large overshoot and a long adjustment time; when T_d is too small, the overshoot is also large, and the corresponding adjustment time will be long. In motion control, the differential action has an important role in suppressing proportional oscillations, so that the robot can make better steering adjustments based on the amount of error variation.

Algorithm 1 PID

```
1: Kp ← suitable_value
2: Ki ← suitable_value
3: Kd ← suitable_value
4: procedure CALOUT()
5:     error ← Dist
6:     time ← current.time
7:     de = error − prev_error
8:     dt = current.time − previous.time
9:     Cp = error
10:    Ci = prev_error + error
11:    Cd = de/dt
12:    prev_error = error
13:    previous.time = current.time
14:    pid_output = Kp ∗ Cp + Ki ∗ Ci + Kd ∗ Cd
15: end procedure
```

Figure 4.12

The biggest feature of the PID controller is that it does not require a specific mathematical model of the controlled object, but the model can be

used to help tune the three parameters of the PID controller. Usually we will use the Z–N method and relay-control method. These two methods can get the parameters with good control effect, but these two methods tend to compete for the first-order inertia plus pure hysteresis system, because our robot is not such a system, we use the experimental test method based on the characteristics of three parameters to tune the PID parameters, and we preferentially adjust the proportional control coefficient, in this time we just reach a state of left and right oscillation, then add differential control, and use the differential time constant to weaken the oscillation effect and smooth turning. and finally the integral control is performed to reduce the time when adjusting the direction of the cat robot after detecting the mouse, so that it immediately faces the direction of movement of the mouse, and the method of normalizing and limiting the duration of the PID action is added,in order to ensure the cat robot will not suck into an error that it cannot turn another direction. To release the PID controller without theoretical perfect parameters, and this has achieved good results in the test.

4.3 Searching

(Maximilian Klingmann)

After covering the functionalities of collision avoidance and the detection of the mouse via the camera we introduced a layer called *Searching* which starts to play a role in the behaviour fusion as soon as the homing informs this layer that the mouse has recently been seed and is now lost. In this case we did not want our robot to simply revert back to randomly looking for the cat but rather use the information we know about the whereabouts of the mouse, that is the last location we detected the mouse in the camera.

In this layer we do not read any sensory data but only two streams of information are fed into the procedure. Firstly we receive the suggested behaviour from the random walk through the forward-feed, that consists of a linear and angular velocity, namely a Twist. Secondly we receive a vector $msg_{bp} = [x, y, w, h, c, s]$ via the back-feed from the homing layer, where x, y, w, h are the x- and y- coordinates as well as the width and height of the last bounding box, where we saw the mouse before we lost sight of it. Furthermore two boolean states are returned with c declaring if a collision is pending and s declaring if we currently see the mouse. As soon as homing reports the mouse is seen a global state is set, so that a search is pending upon losing sight of the mouse.

When losing the mouse a timer is started to track how long it has been since we last saw the mouse. Two callback methods shown in figure 4.13 and figure 4.14 are invoked as soon as information is received via the back-feed and forward-feed respectively.

If the back-feed of homing reports the mouse is not seen anymore and the global state of pending search is set we begin searching for the mouse. When no search is pending and we do not see the mouse we pass through the suggested behaviour received from random walk. This results in the previously explained first prevail gate being used implicitly.

The search method itself takes the last bounding box into account when suggesting a new behaviour. The robot uses a heavily increased angular velocity in the direction the mouse was last seen in. In other words if the

Algorithm 2 Callback of msg_{bp}

```
1: procedure CALLBACK OF msg_bp
2:     if !in_sight then
3:         if SEARCH_PENDING then
4:             search(msg_bp)
5:         end if
6:     else
7:         SEARCH_PENDING ← True
8:         TIME_LAST_SEEN ← 0
9:     end if
10:    Publish msg_bpto Random Walk
11: end procedure
```

Figure 4.13

Algorithm 3 Callback of msg_{fp}

```
1: procedure CALLBACK OF msg_fp
2:     if !SEARCH_PENDING then
3:         Publish msg_fpto Homing (pass through)
4:     end if
5: end procedure
```

Figure 4.14

mouse has recently been seen on the right sight of the camera image the cat robot heavily turns in this direction while still keeping linear velocity expecting to find the mouse as fast as possible again. Time not seeing the mouse increases uncertainty on the knowledge of the mouse's current location, which is why we revert back to random walk after a pre-defined period of time. In our implementation that is achieved by switching off the pending search global state, thus resulting in passing through the suggested behaviour of random walk instead overpowering that input with its own. A pseudo implementation of the search method can be see in Figure 4.16. Furthermore we introduced a time period before the robot should start to turn in the

direction the mouse was lost resulting in the robot driving only straight for that period of time. This time needs to be adjusted depending on the map and complexity of the turf the game is played on. On a field where no obstacles are placed this period of time can be close to zero, as no obstacles can hinder the view. However when running the game on a field with obstacles it might be helpful to keep on driving straight for a couple of ticks, since the mouse could pass an obstacle further back in the area. If the obstacle spans an area on the projected camera image that spans across the center of the image the mouse might be lost left of the images center thus resulting in a left turn when beginning to search. Allowing the cat to keep on driving straight for a couple of ticks however might result in the mouse immediately being found again when it emerges on the other side of the obstacle, i.e. right of the camera's center. In this case the homing algorithm would instantly jump into action again. A precipitated turn without allowing for a linear travel period would in that case result in a suboptimal turning direction, since the cat turns left, the mouse however is further right.

Algorithm 4 Searching

```
1: TURN_DIR = 0
2: procedure SEARCH(msg_bp)
3:     t ← Twist()
4:     t.linear.x = 1
5:     if TIME_BEFORE_SPIN ≤ TIME_LAST_SEEN < TIME_BEFORE_SPIN + TIME_TO_SPIN then
6:         twist.angular.z ← spin()
7:     else
8:         if TIME_LAST_SEEN > TIME_BEFORE_SPIN then
9:             SEARCH_PENDING = False
10:        end if
11:    end if
12:    Publish t to Homing
13: end procedure
```

Figure 4.15

4.4 Random Walk

(Chenyang Zhu)

Random Walk is the first strategy we run, it is aimed at finding the mouse in a short time. so we let it drive forward for a while and scan the whole playgroud in a appropriate speed. And it should not affect the collision avoidance when scanning behaviour turning the robot. And when searching lose finding the mouse, it will come to random walk. it depends how fast we can find the mouse.

In this layer when we get the message from searching, it loses the tracking of the mouse robot. We start our random walk program, in the program we define the variable for the random process, we can control the straight moving time by $TIME\_LINEAR$,and control the turning time by $TIME\_ANGULAR$. We first let the robot move straight for some seconds, and than let the robot turn 360 degrees. In the first, we didn't turn the whole circle, because we thought there would be some collisions in the playground, if there is no collisions in the playground we think turning 360 degrees is faster to find the mouse robot.

We think Random Walk is a decision that should be flexible for the current situation of the playground just like we play the video games, a pure random walk seems to be a good choice, but it also contains a lot of useless moving that wastes time. So what we need to do next is according to the game theory to make a better decision about random walk, but there also exits a very popular method based on game theory called Reinforcement Learning. If we have enough computing power that will be a good choice, because when reinforcement learning is utilized, our cat robot can respond to the environment's situation to improve its performance for catching the mouse robot.

Algorithm 5 Randomwalk

```
1: LIN_TIME = 0
2: TURN_DIR = 0
3: TURNS = 0
4: TIME_LINEAR = 100
5: TIME_ANGULAR = TIME_LINEAR + 20
6: WAYLAYING_TURNS = 1
7: procedure RANDOMWALK()
8:     twist ← Twist()
9:     if LIN_TIME < TIME_LINEAR then
10:        twist.angular.z ← GAS * 0
11:        twist.angular.x ← GLS * 1
12:    else
13:        twist.angular.x ← GLS * 1
14:        if TURN_DIR == 0 then
15:            TURN_DIR = GAS * 3.14
16:            if TURNS == WAYLAYING_TURNS then
17:                TIME_ANGULAR = TIME_LINEAR + 50
18:            end if
19:        else
20:            twist.angular.z ← TURN_DIR
21:        end if
22:    end if
23:    LIN_TIME+ = 1
24: end procedure
```

Figure 4.16

4.5 Conclusion & Future Work

(Julien Stern) After our implementation, we tested the robot in the playgound, adjusted the suitable threshold values for bounding boxes and the three parameters for PID controller. During this field-testing we found that the visual homing task heavily depends on the current lighting conditions, being artificial light in the room or natural light through the windows. This forced us to use a huge range for saturation and value in the HSV color range, in order to correct for different light exposures of the camera. Thus, miss-classified regions in the arena and outside (someone with a reddish T-shirt) were unavoidable.

However, we believe that our goal of creating an autonomous robot that performs its basic tasks with a certain reliability and robustness against environmental changes was reached. Together with tests in simulations that showed wrong odometrical data, this is also the reason why we did not use odometrical data for position calculation or path planning, since these concepts would contradict with our simple and robust philosophy.

Also during the competitive games against mouse robots of other teams, we experienced high signal delay between the robot and the commanding computer, which led to false conclusions during homing and ultimately guiding the robot in wrong directions. This however only appeared against one mouse robot and also never during our single testing. We therefore conclude that the problem was caused by this specific mouse robot and not our implementation. The fact that the other car robot team also reported this phenomenon also suggests this theory.

As future work we consider adding the functionality of detecting the cheese for cat robot, allowing us to, when the cat robot finds the cheese first, spin around it to lure the mouse.

Another point is collision avoidance. Our collision detection is based on the sonar scan data, which limits the precision of the algorithm, because of the number of scan data. Our collision avoidance algorithm would be more robust if we used the laser scan data in the future and fully adapt our algorithm to the VFH algorithm.

4.5.1 Suggestions for future lectures

Explain the arena in detail at the beginning of the course. It can be vital for some parts of the implementation to know beforehand specifics about the environment the robot will operate in.

During presentations we strongly recommend the audience to wait until the end of a presentation before asking questions, giving advice or sharing extra information. Different behaviour interrupts the flow of the presentations and might put presenter into uncomfortable situations which do not help during presentation.

Bibliography

[1] Rodney A. Brooks. A robust layered control syste for a mobile robot. 03 1986.

[2] James L. Crowley. Navigation for an intelligent mobile robot. 1985.

[3] MobileRobots Inc. Pioneer 3 operations manual, 2006.

[4] Javier Minguez, Luis Montano, Thierry Siméon, and Rachid Alami. Global nearness diagram navigation (gnd). volume 1, pages 33 – 39 vol.1, 02 2001.

[5] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[6] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.