TUHH

Technische Universität Hamburg-Harburg

Diploma Thesis

A Multi-Agent Simulation Framework and Control Structure for Collaborative Searching and Tracking

submitted by

Axel Hackbarth

B. Sc. (Hamburg University of Technology), 2007
MEngSt (University of Auckland), 2008

Supervisors:

Professor J. Karl Hedrick (University of California, Berkeley)
Professor Herbert Werner (Hamburg University of Technology)
Professor Edwin Kreuzer (Hamburg University of Technology)

Center for Collaborative Control of Unmanned Vehicles
Department of Mechanical Engineering
University of California, Berkeley

Hamburg, 13th of May 2009

Berkeley, April 30. 2009

I, Axel Hackbarth (Student of Hamburg University of Technology, student ID 29950), hereby declare that this thesis is my own work. Furthermore, I confirm that no other sources have been used than those specified in the thesis itself.

Axel Hackbarth

Abstract

Unmanned aerial vehicles (UAVs) have been widely used in military operations. However, UAVs are becoming increasingly popular for non-military applications. Thus, reliability and robustness must be improved while reducing the cost of testing and implementing these UAV systems. Collaborating teams of UAVs can handle complex tasks with various or changing mission objectives. Simulations provide a low-cost method of testing these systems in a virtual environment.

This paper addresses the general approach of an object-oriented software architecture for the simulation of search and track tasks with multiple vehicles. Based on our simulation environment, a high-level control structure has been developed that demonstrates a good real-time performance. It is an efficient way to solve exploration, searching, and tracking tasks for multiple agents in an unknown multiple moving target environment. The control algorithm uses a state feedback controller to follow a waypoint that is defined by an agent-specific cost function. It navigates the agent on a route to this waypoint by adding the gradient of this cost function at the agents position to the feedback gain, thereby preventing vehicle collisions and avoiding no-fly-zones.

Our simulation environment as well as our control structure are meant to be a starting point for future research. Thus, our work focuses on a broad functionality and usability and demonstrates this by a partly heuristic control algorithm, thereby motivating researchers to contribute to the framework and to develop their own controllers with it.

Acknowledgments

My intellectual and physical journey of tertiary education was an unforgettable experience. I want to thank many people I met on my way: Many that I became very good friends with, and many who supported me because they believed in me. The trust of all these people gave me the strength to walk my way and always kept my schedule filled.

First and foremost I want to thank Professor Kreuzer who taught me the basics of mechanics and gave me the broad knowledge that I rely on every day. I thank him for supervising my Bachelor's Thesis and especially for giving me the possibility to write my Diploma Thesis at UC Berkeley.

I also want to thank Professor Werner for his fabulous lectures in control systems, the inspiring conversations about student exchanges, and for supervising my Diploma Thesis.

Much thanks to Professor Hedrick who invited me to work on his exciting UAV project and gave me a challenging and rewarding topic. He introduced me to his graduate students and he supplied me with all I needed throughout my stay. Seeing the real-world system in action inspired many ideas for my simulation that I hope my simulation will do vice versa.

One person to whom I owe much of my success is Dr. Marc-André Pick, my advisor during my Bachelor's Thesis who helped me to find my style in academic writing and who continues to push and inspire me to this day. He is always the first to know about my academic adventures and has always supported my endeavors. Dr. Pick suggested that I write my Diploma Thesis in Berkeley. Both him and I invested much effort to make this become a reality.

Working at the Vehicle Dynamics Lab (VDL) and the Center for Collaborative Control of Unmanned Vehicles (C3UV) was very rewarding. I made good friends there and had many long discussions, both on and off topic. These awesome labmates dragged me out of the lab to a local pub or put me on a surfboard when I was stuck in my work. They also were there to bring me back to my studies when I was faced with an intellectual drought.

My family and friends have always been a solid foundation to rely on. In particular my father, Manfred, who supported me throughout my whole education with great belief and

even greater freedom to do whatever I felt was good for me. My brother, Felix, has always been a great source of inspiration and information to me, from playing with LEGO™ bricks until studying collaborative control systems. He and his significant other, Britta Werner, and their lovely daughter, Jette, have always had their door opened for me and it felt like a second home to me.

Finally I want to thank Lina, who became a crucial part of my life in a very short time, for making me smile every day in this stressful stage of my thesis. I also want to thank her for being the final reason to come back to this awesome place called the Bay Area.

Contents

List of Figures

Abbreviations and Signs

Abbreviations and Word Definitions

agent	A single mobile sensor which has sensors aboard
target	Moving mission goal that has to be discovered
C3UV	Center for Collaborative Control of Unmanned Vehicles at UC Berkeley
MatLab	A scientific programming environment
OOP	Object-oriented programming, a structured method for software design
SVN	Subversion, a version control system
UAV	Unmanned aerial vehicle

General Notation

$\mathbf{r}$	Position vector
$\mathbf{x}$	State vector
t	Global time variable
Δt	Time step of constant length
$\mathbf{0}$	A zero matrix or zero vector of respective size
$\mathbf{A}^{-1}$	The inverse matrix of $\mathbf{A}$
$\mathbf{I}$	An identity matrix of respective size
${}_b\mathbf{R}_a$	Conversion matrix from coordinate system a to coordinate system b

Latin Symbols

m	Mass value
t	Time value
r	Spatial value
$\mathbf{r}$	Spatial vector

$\mathbf{u}$	Input vector
$\mathbf{x}$	State vector
$\mathbf{C}$	Cost map
$\mathbf{K}$	Knowledge map

Operators

Δ	Difference from minimum to maximum or from one step to the next
∇	The derivative in multi-dimensional space: $\nabla = \left[\frac{\partial}{\partial x} \quad \frac{\partial}{\partial y} + \quad \frac{\partial}{\partial z}\right]^T$
$\mathrm{Var}\, x$	Variance of x

Indices

k	Time-discrete variable at time $t = k\Delta t$

Coordinate Systems

x-axis	Normally directed to right
y-axis	Normally directed to forward
z-axis	Normally directed to up

1. Introduction

Autonomous aerial vehicles have the potential to protect humans from dangers that can be observed and predicted from above, but are too dangerous, too complex, or simply too expensive for human operators. Technological advances in and decreasing costs of the components for autonomous vehicles are leading to a strongly increasing interest in mobile sensor systems. Besides military applications, an increasing number of civil projects make use of this technology [Hof08]. Mobile aerial sensor teams are considered for fire patrol, mission planning in conflict or disaster areas, emergency surf and mountain rescue, and others.

These new fields of application come with new challenges. The fundamentals of autonomous vehicles have been overcome on a high cost which clears the way for many interesting projects with a limited budget. Therefore, the technology has to be affordable and off-the-shelf components have to be integrated into systems to work well together. Furthermore, civil airspace has much higher security standards which enforces a very robust and reliable system.

To satisfy all the requirements in this new environment, robust functionality has to be proven by simulations. They should be a substantial part throughout the whole design process and adapt to the changes and the complexity of each specific mission.

1.1. State of the Art

Autonomous vehicles are used in different environments to fulfill tasks that are too dangerous or too expensive to deploy human operators. Nevertheless, most current autonomous vehicles have a very low degree of autonomy and require humans in the loop to make high level decisions. Current research focuses on improving the degree of autonomy, as well as the robustness, while increasing the functionality at the same time. An optimal control input estimated over an infinite horizon would be the best possible control strategy in terms of performance if the cost of computation is neglected. For a realizable and efficient algorithm the receding horizon has to be limited.

The founder of modern information theory was Claude E. Shannon who defined the term entropy for the first time in his publication 'A mathematical theory of communication' [Sha48]. Based on this context, information-theoretic controllers have been designed to minimize the entropy of a system. In the context of a search problem with a constant rate of observations, an entropy minimizing control can be shown to be equivalent to minimizing the expected number of future observations required to locate the target [RTG+07; Rya08]. These control-structures find optimal routes over the predicted horizon, but the Entropy calculation is computationally very expensive and not yet feasible in real-time systems. At ARC Centre of Excellence for Autonomous Systems, recursive Bayesian filtering is applied to the search-and-tracking of multiple targets with heterogeneous vehicles [FBLD06; LFW08] Also at ARC, a decentralized information gathering control architecture has been developed and tested in a real system [CSG+06; CGS08].

At Stanford, the experience gained in the DARPA challenge for fully autonomous driven off-road vehicles [TMD+06] have been transfered to the control of mobile sensors that are used to collaboratively search with highly nonlinear sensors [HWT06]. In [BG08], a mixed-initiative UAV-assisted search and rescue framework has been developed. Their focus lies in collaborative assignment of search areas by automated agents and human operators. Kuwata compared and extended various different receding horizon controllers for path planning of single and multiple UAV missions against each other [Kuw07].

Particle filtering is a Sequential Monte Carlo method and uses a random set of weighted particles which describe the form of the probability density function (PDF) [KBD03]. These particles will be processed by recursive Bayes filtering to get the updated PDF. For a robust performance, the particles have to be resampled and their weights adjusted to avoid an accumulation of the particles.

Grid-based methods have been used with virtual attractive and repulsive potential fields [NCT+05].

1.2. Contribution

Highly integrated control structures are optimized to perform well, but only in a very specific scenario. Changes to the system can require a complete redesign of the control algorithms.

With a modular software architecture new technologies can be implemented with very little effort. Furthermore, control development for teams of UAVs is, due to the complexity of these systems, much better handled in collaboration with other researchers. This moti-

vated the development for an object-oriented simulation environment that is designed for multi-agent multi-target searching and tracking missions. Due to the encapsulated structure, contributions can be easily added to the system. Researchers can adapt to and work on the project with a very low learning curve and get a deeper understanding of specific functions over time. Code collaboration is managed by a version control system to allow different researchers to work on the same project at the same time, without overwriting each others achievements.

This simulation environment models all motion with a second order linear motion model. The targets' positions are estimated by a Kalman Filter approach and updated by a noise-affected camera model. The exploration grade of the region is saved as a mission state and a time-dependent model has been developed.

Furthermore, we developed an innovative control structure that is based on a global map of mission accomplishment. This map is manipulated to represent a cost map specifically for each agent. This agent-specific map includes other agents, their waypoints, no-fly-zones and the travel time cost. Based on the minimum of that map, the agent determines its waypoint and follows the local gradient towards this waypoint.

2. Real World System Infrastructure

Simulations are means to develop for, analyze, and recreate real world scenarios. Thus, every simulation is meant to represent the real world in some form of abstraction level. The real world system is therefore the system that we have to analyze before we start modeling it.

Unmanned aerial vehicles (UAVs) are used in teams for various applications which have the common ground of distributed mobile surveillance or measurement. Thus, UAVs are equipped with at least one sensor. For solving tasks collaboratively, the agents have to communicate through a communication device to each other and a ground station. For the autonomous flight, a UAV has to include a low-level autopilot that controls the airplane actuators and allows simple waypoint following tasks. Furthermore, high-level control and sensor evaluation demand for a reasonable on-board computation.

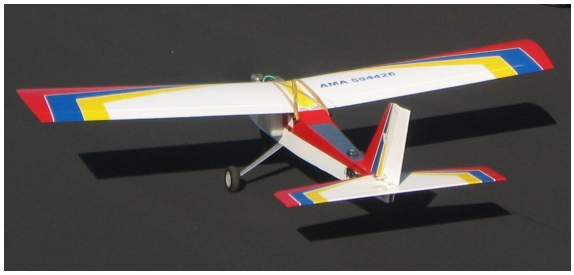
The following sections describe the current state and setup of the fleet of the Center for Collaborative Control of Unmanned Vehicles (C3UV) at UC Berkeley.

2.1. Airframes

A wide variety of airframes for UAVs are commercially available. The choice of size, payload and propulsion is a question of the intended use. Research at C3UV goes into many different directions, but a major goal is to collaboratively accomplish tasks with a heterogeneous fleet of UAVs. Four different airframes are currently in use (Fig. 2.1).

2.1.1. Rascal

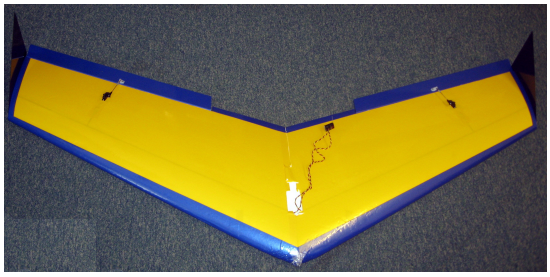
The Rascal airframe is a reliable remote controlled airplane intended for hobby use. It was reinforced to carry an increased payload for sensors, autopilot, and extra gas tanks for an increased flight time. It is powered by a two-stroke engine and can take off with an additional weight of up to 3 kg . This leaves enough room for the PC104 computation unit, the Piccolo autopilot and a fixed-wing camera.



(a) Trainer

(b) Rascal

(c) Bat



(d) Zagi

Figure 2.1.: The four airframes have very different characteristics and are used for different mission objectives.

2.1.2. Bat

The Bat 4 by MLB Company is an airframe designed specifically for autonomous UAV missions. It comes equipped with a gimbaled camera and has a big payload tray for on-board computation and sensory equipment. Its flight time with up to 8 hours, depending on the additional weight, allows for long lasting missions.

2.1.3. Trainer

The Trainer airframe (figure 2.1(a)) is used as a remote controlled airplane for teaching purposes and as a manually controlled sensor node. Its limited size and take-off weight does not permit our standard payload tray, but research is done in using integrating computing environments, such as in todays smartphones, to make this unit fully autonomous. It can be equipped with the Piccolo autopilot to get real time data and act as an aerial target for other UAVs.

	Bat	Rascal	Zagi
Max take-off weight	43 kg	9.7 kg	8 kg
Empty weight	25 kg	8.1 kg	7 kg
Max fuel weight	18 kg	1.6 kg	1 kg
Cruise speed	28 $\frac{m}{s}$	25 $\frac{m}{s}$	18 $\frac{m}{s}$
Maximum speed	33 $\frac{m}{s}$		
Maximum flight time	8 h	2 h	0.5 h
Wingspan	3.81 m	2.79 m	1.83 m
Engine type	106cc two-stroke	32cc two-stroke	electro
Engine power	7681 W	2638 W	
Construction	kevlar composite	balsa & plywood	styrofoam
Price	35000$	1200$	200$

Table 2.1.: The heterogeneous fleet includes airframes for different applications that we intend to use collaboratively to benefit from each agents' capabilities.

2.1.4. Zagi

The Zagi is a new development based on Styrofoam delta wings. It has a flight time of only 0.5 hours when used with the engine, but due to its good performance as a glider it can be used for special scenarios with thermal winds or in upwind zones. For these scenarios, an unlimited flight time is theoretically possible. The power needed for the servos, communication, and on-board computation can then be generated by a wind turbine.

2.2. Sensors and Actuators

Sensors are needed as an interface between the real world and our artificial models. Which sensors are necessary depends on the mission objective. Our system model (Chapter 2) gives us all states that we want modeled and observed in our system. Where some sensors are necessary to be able to observe states, others might be added for a better performance.

The different airframes allow for different kinds and amounts of sensors. All units are equipped with a Global Positioning System (GPS) and an inertial measurement unit (IMU). The GPS sensor measures an absolute position state, whereas the IMU detects longitudinal accelerations and rotational velocities. Furthermore, a Pitot-Static system is needed for detecting the flight speed through the air, which can differ from the original system due to wind. All these sensor informations are processed in the on-board Piccolo

autopilot from Cloudcap Technologies. The autopilot is also controlling the servo motors for the ailerons, the rudder, the flaps, and the elevators of the airplane.

Directly for the mission execution, sensors are needed that observe the mission state and can detect mission objectives. This can be done by communicating the states to our units, but these mission states have to be detected in some way, either by the unit itself, other aerial units, satellites, or ground units.

For many objectives, a camera is a good universal sensor when used with image processing and target detection algorithms. An additional infrared camera adds especially for night missions very distinct detection information. The Bat airframe (Section 2.1.2) comes equipped with a gimbaled camera, which is servo-actuated and has three rotational degrees of freedom to follow a target independently from the airframe's orientation. For smaller systems, like the Rascal airframe (Section 2.1.1), the extra weight for a gimbal mount cannot be carried and our camera is simply downwards pointing. This leads to a sensor footprint that is dependent on the bank angle of our airplane.

2.3. High-Level Computation Unit

High-level control for multiple vehicle scenarios has a high demand for on-board computation. A QNX real-time Linux system which runs on a PC-104 has been integrated into the payload.

PC-104 is an embedded computer standard which allows the integration of flexible and interchangeable hardware configurations in a small form factor. The main hardware components are PC-104 cards for the processor and memory, an input and output interface, a wireless network card based on 802.11b, and a framegrabber to do on-board real-time image processing.

The software that runs on the PC-104 is a QNX real-time Linux. The architecture is being redesigned to incorporate the Orca system toolbox for robotics and ICE middleware for communication handling with sensors and actuators. This new framework allows easier component integration because many problems of interfacing and communication are already integrated. The high-level control algorithms and image processing software are designed in C++ and integrated into the Orca / ICE framework.

3. Modeling the System

Modeling the real world system as described in Chapter 2 is an important step towards a simulation environment. The interaction and collaboration of a team of UAVs is the focus of our modeling process (Figure 3.1). Therefore, the UAVs themselves (hereafter called agents) have to be modeled and the region in which they move has to be defined. The tasks that should be accomplished can be to explore and measure the region, or to discover and / or track standing or moving objects (hereafter called targets). Similarly, the targets must be modeled in order to simulate the targets' motion behavior. The sensors of the agents and the communication between the simulation objects is also a major task in the design of the simulation.

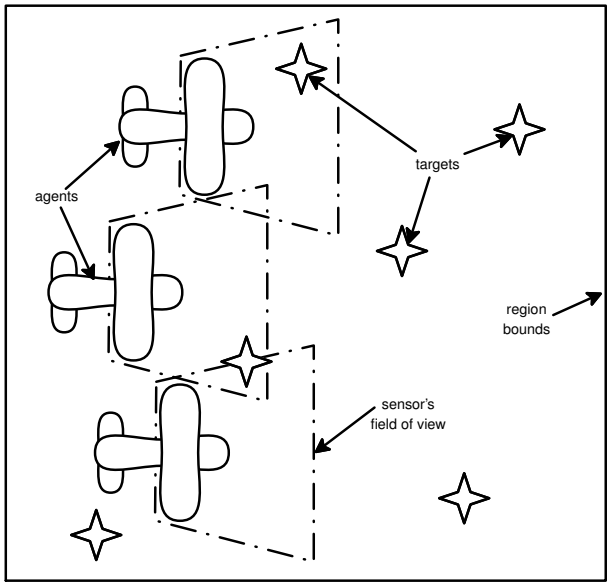


Figure 3.1.: The system models how all objects act and interact that are involved in the simulation.

For high level control simulation purposes the real system does not have to be modeled in its full dynamics. Decreasing the states of the whole system to a working minimum results in a model that is much easier to understand, implement, and compute. Consequently, there are less errors during the software and controller development.

Once the simplified simulation model has proven to run smoothly, more complex dynamic models can be considered. This approach asks for a dynamic simulation environment that grows with and can adapt to its tasks. The long term goal is to design a simulation environment that can model each component of the environment in a predefined complexity. Thus, control algorithms can be designed and tested in an early stage and improved while increasing the complexity of the simulation.

This concept demands for a high amount of flexibility in the software architecture which is further discussed in Chapter 5.

3.1. Space

Space in the real world consists of three dimensions with a specified measurement size. The simulation space is a simplified representation and has been cut down to two dimensions parallel to the ground. The measurement unit throughout the whole simulation meter.

Space related measures are stored in double precision without any further discretization. In cases where values for a whole area are stored or computed, the region space is discretized in a grid to bound memory and computational needs. Our environment supports multiple regions in different grid resolutions and different positions.

The simulations were successfully tested with grids ranging in sizes from 16×10 up to 1600×1000 , with the latter having a computation time of around 30 seconds depending on the other parameters. Simulations done on a 160×100 grid were found to be a good balance between computation time and grid resolution.

3.2. Time

The simulation is running in a discretized time environment. Each time step has a constant length which is defined before the simulation starts. All simulation objects perform their tasks (e.g. moving) which are scheduled and triggered by the global simulation time. Using this global constant time step gives the whole simulation a clock cycle and simplifies the execution of objects.

When this system is ready to be implemented in the real system (Chapter 2), the latter condition is not easily feasible and the system has to be extended to allow asynchronous data handling.

For our simulation, a time step of $\Delta t = 1s$ was a good compromise between near real time performance (depending on other settings) and time resolution.

3.3. Moving Objects

All vehicles in the real world system are represented as moving objects in the simulation. The moving object model includes a general form of vehicle dynamics and limiting properties.

A vehicle can be modeled to have very basic kinematics but can also incorporate higher order dynamics. Low level controllers take these dynamics into account, whereas for high level control these higher order system dynamics are generally negligible (Figure 3.2).

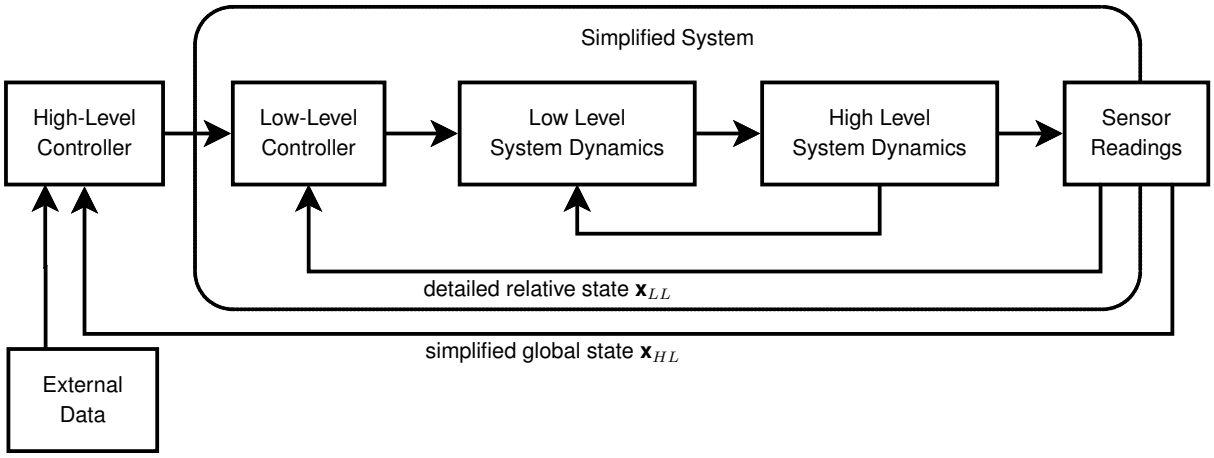


Figure 3.2.: The higher order low-level aircraft dynamics are controlled by the low-level controller and simplify the high-level system.

3.3.1. Spatial Degrees of Freedom

Aerial vehicles can move in a three dimensional space. Adding the orientation of the airplane results in three more rotational degrees of freedom. The position and orientation of the rigid vehicle can be uniquely described by the state vector:

$$\mathbf{r} = \begin{bmatrix} x & y & z & \phi & \theta & \psi \end{bmatrix}^T . \tag{3.1}$$

Assuming that each vehicle moves only along a 2D plane parallel to the ground, the degrees of freedom are reduced by the two rotational degrees ϕ and θ, and the one

translational degree z to:

$$\mathbf{r} = \begin{bmatrix} x & y & \psi \end{bmatrix}^T \ . \tag{3.2}$$

To further reduce the complexity, moving objects are modeled as points without an orientation. This leaves only two translational degrees of freedom:

$$\mathbf{r} = \begin{bmatrix} x & y \end{bmatrix}^T \ . \tag{3.3}$$

3.3.2. Time-Dependence and Dynamics

A real vehicle makes neither instantaneous changes to its position, nor to the velocity with respect to time. The engine dynamics are assumed to be much more responsive than the inertia dynamics of the system and are therefore neglected. For a point mass the kinematic equation can be derived with Newton's Law and the control force $\mathbf{u}$

$$m\ddot{\mathbf{r}} = \mathbf{u} \ . \tag{3.4}$$

Integrating the latter equation over time results in the change of velocity, a second integration results in the change of position. Thus, the velocity is part of the system state $\mathbf{x}$:

$$\mathbf{x} = \begin{bmatrix} \mathbf{r} & \dot{\mathbf{r}} \end{bmatrix}^T = \begin{bmatrix} x & y & \dot{x} & \dot{y} \end{bmatrix}^T \ . \tag{3.5}$$

In a state-space representation the system dynamics are a combination of the state (equation 3.5) and the kinematic equation 3.4:

$$\frac{d}{dt}\begin{pmatrix} \mathbf{r} \\ \dot{\mathbf{r}} \end{pmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}\begin{pmatrix} \mathbf{r} \\ \dot{\mathbf{r}} \end{pmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{I}/m \end{bmatrix}\mathbf{u} \ . \tag{3.6}$$

3.3.3. Discretization

The discrete time environment (section 3.2) requires the state equations to be discretized. The general form of a discrete system is:

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k \ . \tag{3.7}$$

For a linear system, the latter equation can be rewritten in matrix notation as:

$$\mathbf{x}_k = \mathbf{F}_k\mathbf{x}_{k-1} + \mathbf{B}_k\mathbf{u}_k + \mathbf{w}_k \ . \tag{3.8}$$

A Taylor series of the position state (equation 3.6) developed at time $t_k = t$ for time $t_{k-1} = t - \Delta t$ gives us the discretized system equations of our moving object:

$$\begin{pmatrix} \mathbf{r}_k \\ \dot{\mathbf{r}}_k \end{pmatrix} = \begin{bmatrix} \mathbf{I} & \Delta t\mathbf{I} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{pmatrix} \mathbf{r}_{k-1} \\ \dot{\mathbf{r}}_{k-1} \end{pmatrix} + \begin{bmatrix} \frac{-\Delta t^2}{2m}\mathbf{I} \\ \frac{\Delta t}{m}\mathbf{I} \end{bmatrix} \mathbf{u}_k \ , \tag{3.9}$$

and developed at time $t_{k-1} = t - \Delta t$ for time $t_k = t$

$$\begin{pmatrix} \mathbf{r}_k \\ \dot{\mathbf{r}}_k \end{pmatrix} = \begin{bmatrix} \mathbf{I} & \Delta t\mathbf{I} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{pmatrix} \mathbf{r}_{k-1} \\ \dot{\mathbf{r}}_{k-1} \end{pmatrix} + \begin{bmatrix} \frac{\Delta t^2}{2m}\mathbf{I} \\ \frac{\Delta t}{m}\mathbf{I} \end{bmatrix} \mathbf{u}_{k-1} \ . \tag{3.10}$$

With the latter equations the system dynamics are modeled in a simplified way while still allowing motion in the two lateral dimensions that are of most interest to us.

The major drawback of this model is the lack of orientation information. Adding these results in the unicycle model with a state vector that includes the orientation and the rate of change of orientation. Perpendicular to the orientation, the velocity is always zero, thus, the state vector for the unicycle model is

$$\mathbf{x} = \begin{bmatrix} x & y & \psi & v & \dot{\psi} \end{bmatrix}^T \ . \tag{3.11}$$

The dependency of position and velocity in this model is defined by the trigonometric functions and thereby introduces nonlinearities.

3.3.4. Limits

The moving object has several limitations on some regions which make the system nonlinear. The input is bounded by a maximum acceleration or a maximum force. Every time the input is set, it is controlled if the norm of the acceleration does not exceed a set limit. Similarly, the position and velocity are checked for their limits. When the position bounds are reached and the `BounceOffWalls` property is set, the velocity component perpendicular to the wall is being inverted. If the `StopAtWall` property is set, this velocity component is set to 0 . In case nothing is set, the moving object can travel independently from the region limits.

3.4. Targets

Targets are objects in the environment that are of interest to the mission goal and can be detected by the sensors. They are based on moving objects as described in the latter part of section 3.3. These targets are assumed to stand still and move on a 2D layer.

Sensors detect targets by pointing at the 2D layer. The targets can be programmed to follow specific routes or move along randomly in the region. To prevent them from moving out of region, they can be set to bounce off walls like a ball hitting the curb in a game of billiards.

The targets are only known to the simulation environment and the targets themselves. This restriction is necessary to simulate the real world situation where targets are not known to the agents.

For the current state of the simulation environment, it is assumed that each target has a unique identifier and can be distinguished from other targets. A data association filter is planned for a future release of our software and will be capable of applying different methods of association. The nearest neighbor approach is a simple algorithm to map the found target to the nearest estimated target. A joint probabilistic data association filter (JPDAF) takes the probabilistic theory into account by weighting the covariances of the target state estimates accordingly. To be able to detect new targets, a probability of appearance has to be considered and compared with the covariance of the target state estimate.

3.5. Agents

Unmanned aerial vehicles are generalized as agents with sensors. These agents' positions and velocities are the controllable states of the real world environment. Combining the agents with sensors creates the means to achieve a mission goal.

Agents in our simulation are based on moving objects (Section 3.3) and operate on a 2D layer above the targets' layer. They can be equipped with one or more sensors and include a controller which communicates with other agents for task allocation and data fusion.

3.6. Sensors

In the simulation environment sensors are attached to agents and represent downwards pointing cameras. These sensors are the means to detect targets in the field of view and communicate their findings to the controller. The sensor model includes a finite field of view representing the bounds of the camera frame. In the case that a target is inside the field of view while a sensor takes a measurement, noise is added to the position of this target. The uncertainty model is a rough estimation of digital camera's aberration errors defined by the discretization through the sensor and a lens aberration which is modeled

to increase quadratically with the distance from the optical center:

$$\mathrm{Var}_{sensor}(\mathbf{r}) = \sigma^2_{discretization} + \sigma^2_{lens} \left\| \mathbf{r} - \mathbf{r}_{center} \right\|^2 \ . \tag{3.12}$$

The sensor model also includes the image processing steps that are necessary to detect a target. For our simulation we assume perfect image processing so that we can always detect the target with the accuracy of our sensor system.

3.7. Region Estimation

The region exploration $\mathbf{K}$ is the state of knowledge that we have about our region at each discretized position. This knowledge increases with every new measurement at the position of each sensor's footprint at that time on the map. Consequently, the knowledge gain has been modeled as the inverse of the sensor's variance map (equation 3.12):

$$\mathbf{K}_{k,gain} = \mathrm{Var}^{-1}_{sensor} \ . \tag{3.13}$$

The true time dependency is very complex and mainly depends on how fast and random we assume the motion of the targets. For static targets, the knowledge of an already visited position does not change with time because no target would move to this point. If we assume the targets to be in motion, the knowledge of the map blurs with the surrounding region over time since targets can move from unknown regions into the known region. This is modeled with the convolution integral and a Gaussian kernel $\mathbf{G}$.

$$\mathbf{K}_{blur} = \sum_{m \in X} \sum_{n \in Y} \mathbf{G}(m,n) \cdot \mathbf{K}(x-m, y-n) \ . \tag{3.14}$$

Also, the knowledge of the region decreases over time with the fading factor α_{fading} .

From equations 3.13 and 3.14 follows:

$$\mathbf{K}_k = \alpha_{fading} \mathbf{K}_{k-1,blur} + \sum_{sensors} \mathrm{Var}^{-1}_{sensor} \ . \tag{3.15}$$

3.8. Target State Estimation

As described in section 3, the discrete system can be used to model the system behavior. Since the motion of the vehicles is not known when not in any sensors' field of view, it has to be estimated. A Kalman filter is a recursive filter that estimates the state of a linear Markov process.

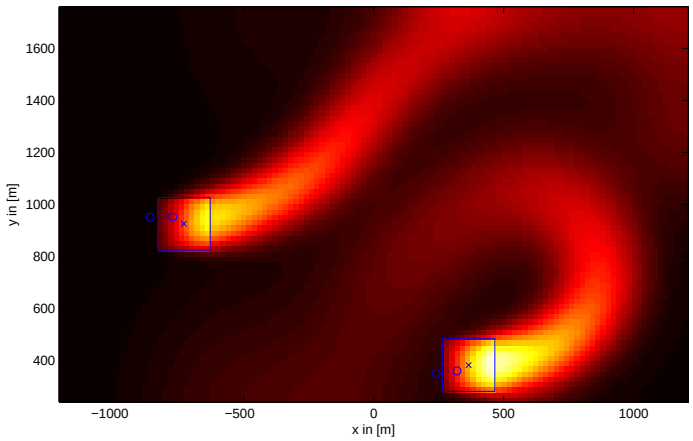


Figure 3.3.: The source of the region exploration are the sensor footprints of cameras which accumulate, but darken and blur over time.

3.8.1. Predict

For the prediction of the target estimate, the same model as for the target is used.

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}_k \hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B}_{k-1}\mathbf{u}_{k-1} \tag{3.16}$$

Consequently, with the same initial values and no noise disturbance, the target and target estimate will always correlate.

We assume a zero-mean white Gaussian distributed noise input for the target state and the camera error (see equation 3.12). This leads to a discrepancy in the target state and its estimate (Fig. 3.5).

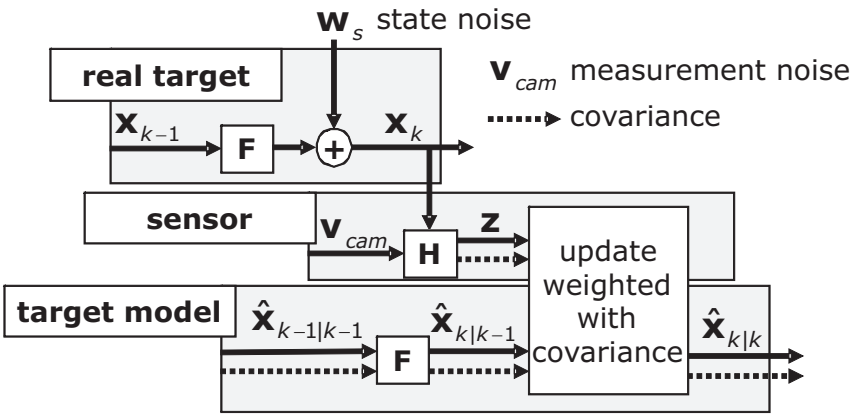


Figure 3.4.: The Kalman filter estimates the target's state when it is not detected and updates it from the noise camera measurement.

To predict the growth of uncertainty of this estimate with time, the estimate covariance matrix has to be processed by the system model and the state noise covariance of the real model has to be added:

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1|k-1} \mathbf{F}_k^{\mathrm{T}} + \mathbf{Q}_{k-1} \ . \tag{3.17}$$

3.8.2. Update

If a sensor measurement has detected a target, the state estimate has to be updated. The update is based on the covariance weighted average of the measured value and the state estimate. This means that a measurement with a very low uncertainty corrects the state estimate drastically (see Fig. 3.5, whereas for a high uncertainty (higher than for the state estimate) the state estimate stays dominant and is only slightly corrected. The

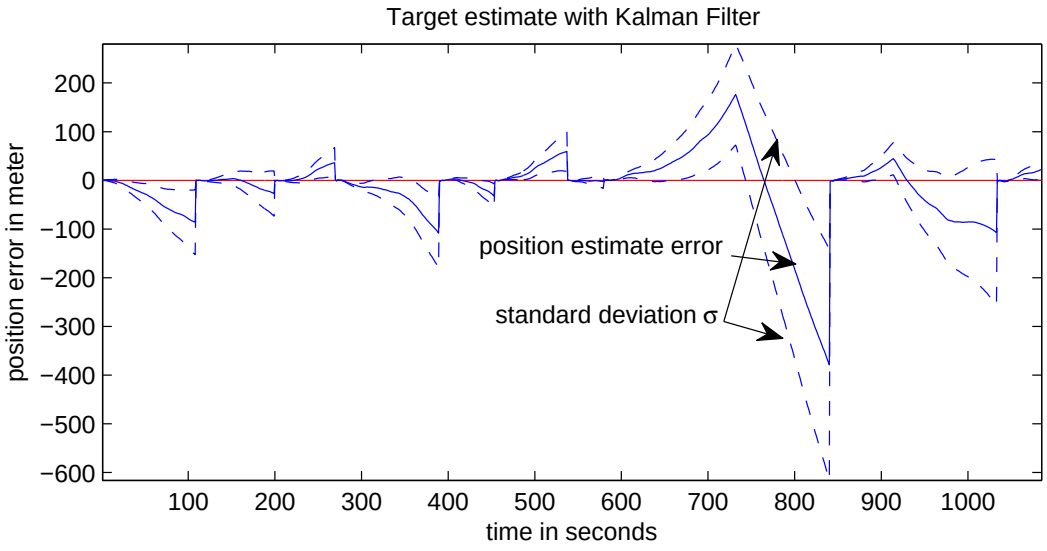


Figure 3.5.: The update due to the camera measurement reduces the position uncertainty drastically because of the low measurement noise.

measurement residual is defined as

$$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_{k|k-1} \ . \tag{3.18}$$

The covariance residual is similarly defined as

$$\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^{\mathrm{T}} + \mathbf{R}_k \ . \tag{3.19}$$

The optimal Kalman gain $\mathbf{K}_k$ is calculated from the target's predicted covariance (equation 3.17 and the covariance residual:

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^{\mathrm{T}} \mathbf{S}_k^{-1} \ . \tag{3.20}$$

The updated state estimate is the predicted state estimate and the measurement residual scaled by the optimal Kalman gain:

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \tilde{\mathbf{y}}_k \ . \tag{3.21}$$

Similarly, the covariance is defined:

$$\mathbf{P}_{k|k} = (I - \mathbf{K}_k \mathbf{H}_k)\mathbf{P}_{k|k-1} \ . \tag{3.22}$$

4. Mission Control Strategy

Our mission control is a high level control algorithm that coordinates a team of agents to achieve an abstract list of tasks. The tasks can be divided into primary tasks that are defined by the mission objective, and secondary tasks that are the restrictions necessary for a seamlessly executed mission.

Our main mission task is to search and track moving targets in a bounded area. This problem can be stated as a single probabilistic problem as done in [Rya08] and [Tis08]. In the first project the author implemented a probabilistically correct algorithm to predict the entropy over a receding horizon. The research showed that entropy calculation is, computationally, a very expensive task, even when searching for a single target unfeasible in a real-time application aboard a UAV.

Furthermore, an optimal control algorithm is always based on a specific measure and a specific time. Optimal control is hard to define, particularly for heterogeneous tasks (e.g. maintaining connectivity, searching for new targets, and tracking found targets while avoiding no-fly-zones) due to the subjective importance of each task. This drives us to develop a non-optimal control that attempts to model these consequences in a simplified way while facilitating better computational performance.

For non-static mission goals, the control strategy has to be able to react dynamically to changing mission parameters. Pre-planned input sequences cannot be applied and the inputs have to be computed in real time. This can be done on a ground station that gathers all data from the mobile sensor system or, for a higher degree of autonomy, by a decentralized control structure that runs on each agent separately. Our object-based control design allows both implementations, with the decentralized control being more challenging due to the communication issues.

4.1. Control Structure

Our control structure consists of a cost-map-based high-level controller that chooses a waypoint based on the agent specific global minimum cost for each agent. The cost map

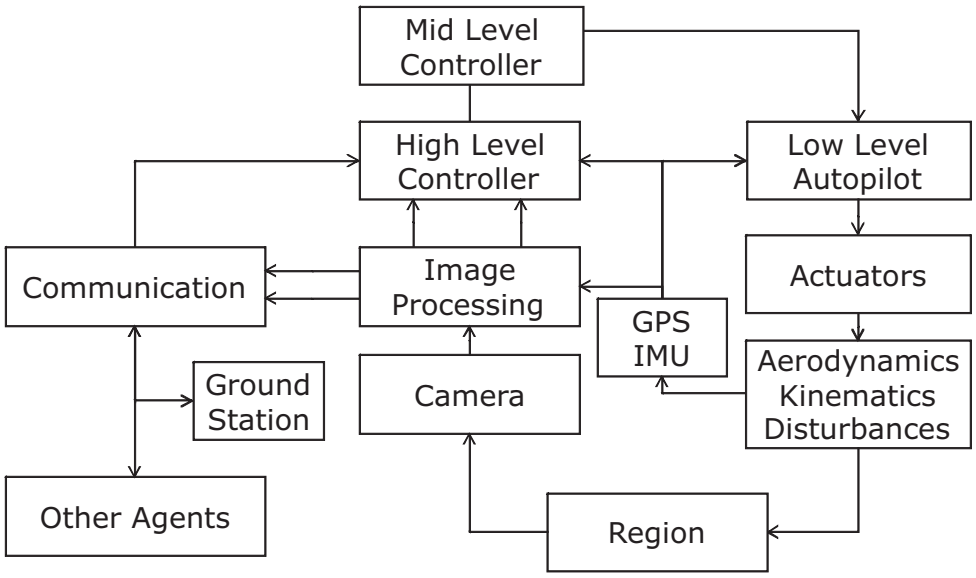


Figure 4.1.: Our control loop is split up into different control levels, a high and a mid level controller. The mid-level controller is a waypoint follower which is fed by the high level controller, a decision-making process for our mission objective.

in particular is a significant step towards our mission goal in a specific region. This means acquiring many measurements of the specific region or target results in high costs for the agent.

Superimposed on this map, which is the same for all agents, are agent-specific costs. These include other agents' position, other agents' waypoints, and the travel time to reach a specific position.

Given the chosen waypoint, a state feedback controller generates the input signal for smooth waypoint following.

$$\mathbf{u}_{agent,waypoint} = K(\mathbf{r}_{waypoint} - \mathbf{r}_{agent} - K_{vel}\dot{\mathbf{r}}_{agent}) \ . \tag{4.1}$$

We expect an increased performance especially for tracking moving targets if the waypoint state is modeled with a velocity.

$$\mathbf{u}_{agent,waypoint} = K(\mathbf{r}_{waypoint} - \mathbf{r}_{agent} + K_{vel}(\dot{\mathbf{r}}_{waypoint} - \dot{\mathbf{r}}_{agent}) \ . \tag{4.2}$$

The described method allows us to follow waypoints with a decent performance, but the path the agent takes might lead through regions with a high cost, for example if another agent is on that path. The cost for going through that region is very high because it might result in a collision of these two agents, but at least in a redundant sensor mapping. To avoid the agents to go through high cost paths and find a route with a low overall cost,

the negative local gradient at the agents position is added to the control input:

$$\mathbf{u}_{agent,gradient} = -K_{grad}\nabla J(\mathbf{r}_{agent}) = K_{grad}\begin{bmatrix}\frac{\partial J(\mathbf{r}_{agent})}{\partial x}\\ \frac{\partial J(\mathbf{r}_{agent})}{\partial y}\end{bmatrix} . \tag{4.3}$$

Equations 4.1 and 4.3 are combined to form the agent input:

$$\mathbf{u}_{agent} = \mathbf{u}_{agent,waypoint} + \mathbf{u}_{agent,gradient} . \tag{4.4}$$

With this control formulation we were able to achieve a good performance for our agent with bounded acceleration / input and bounded velocity with the following values:

$$K = 0.4 , \tag{4.5}$$

$$K_{vel} = 2 , \tag{4.6}$$

$$K_{grad} = 2000 . \tag{4.7}$$

These values might need further tuning and optimization, but for a first guess these values show a good behavior, as can be seen in Figure 4.2.

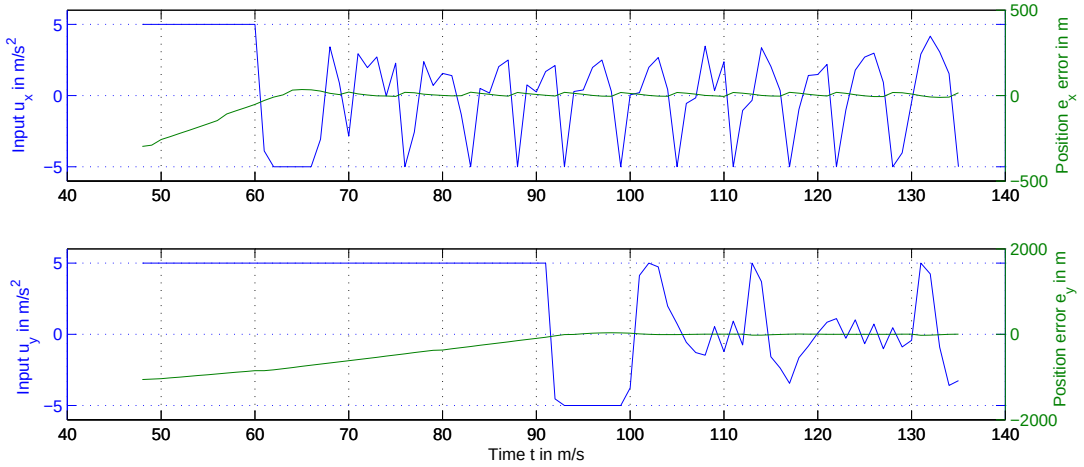


Figure 4.2.: The state feedback controller fulfills its task but parameter tuning is necessary for a better performance.

4.2. Cost Potentials

As pointed out in the latter section, the designed controller is based on a cost grid map. The cost is defined from an agent's perspective. Consequently, no fly zones and regions that have been well explored have a high cost so that the agent avoids it. Unexplored

regions and targets with a high uncertainty have a very low cost, thereby attracting agents. In [NCT+05] these so-called virtual repulsive and attractive potentials have been applied for cooperative guidance control of ground vehicles.

Furthermore, the different cost maps can be categorized into global maps that are valid for all agents and local maps which are only valid for a specific agent. The global maps document the mission achievements, which are the level of exploration of the region, and the uncertainty of the targets.

The exploration of the region is exactly the region knowledge which has been defined in equation 3.15 (Section 3.7).

$$\mathbf{C}_{region} = \mathbf{K}_{region} \ . \tag{4.8}$$

The region exploration cost map is illustrated in Figure 4.3(c).

The uncertainty of the targets is the other agent invariant cost map. The target state estimate and its covariance have been modeled in Section 3.8. The cost map consists of a negative two-dimensional Gaussian distributions for each target. The standard deviation for the Gaussian distribution $\mathbf{G}$ is the standard deviation σ of the position estimate, whereas the amplitude of the Gaussian is derived from the mean value of the standard deviation of the full state estimate.

$$\mathbf{C}_{targets} = \sum_{targets} \sqrt{\left\| \sigma_{targetstate} \right\|} \mathbf{G}_{targetposition} \ . \tag{4.9}$$

The target uncertainty cost map is illustrated in Figure 4.3(d).

These two latter cost maps combined represent the global mission achievement:

$$\mathbf{C}_{global} = \mathbf{C}_{region} + \mathbf{C}_{targets} \ . \tag{4.10}$$

The mission achievement map is illustrated in Figure 4.3(b). The mean value of this map is plotted as a time history to supervise the system performance over time.

The local agent-specific maps are based on a specific agent, that is mostly excluded in the following cost maps.

To avoid collisions with other agents, they are modeled as repulsive $1/x^2$ function with the sensor's field of view as the scaling factor for the x and y coordinate:

$$\mathbf{C}_{agents}(\mathbf{r}) = \sum_{agents \neq this} \left\| \Delta\mathbf{r}_{FieldOfView}(\mathbf{r} - \mathbf{r_a}) \right\|^{-2} \ . \tag{4.11}$$

The agent repulsion map is illustrated in Figure 4.3(e).

To avoid redundant waypoint mapping, the other agents' waypoints are modeled as a Gaussians:

$$\mathbf{C}_{waypoints}(\mathbf{r}) = \sum_{agents \neq this} \mathbf{G}(\sigma = \Delta\mathbf{r}_{FieldOfView}) \ . \tag{4.12}$$

The waypoint repulsion map is illustrated in Figure 4.3(g).

Furthermore, the time to travel to a specific point has been modeled by a squared potential:

$$\mathbf{C}_{time}(x, y) = \left\| \mathbf{r} \right\|^2 \ . \tag{4.13}$$

The travel time cost map is illustrated in Figure 4.3(f).

The local agent's cost map is the sum of all latter maps, weighted with specific variable weights:

$$\mathbf{C}_{local} = \sum_{cost} \alpha_i \mathbf{C}_i \ . \tag{4.14}$$

The complete cost map is illustrated in Figure 4.3(h).

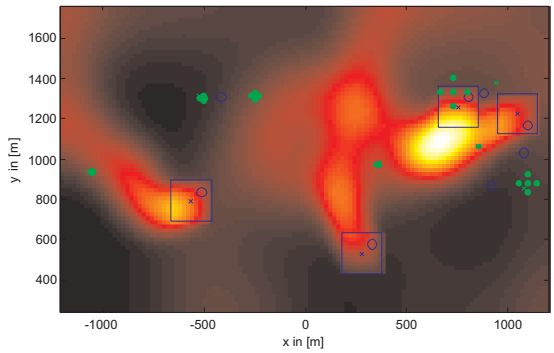
Additional maps can easily be included, but with a high number of cost functions the mission objective might become unclear and result in wrong decisions.

4.3. Prediction

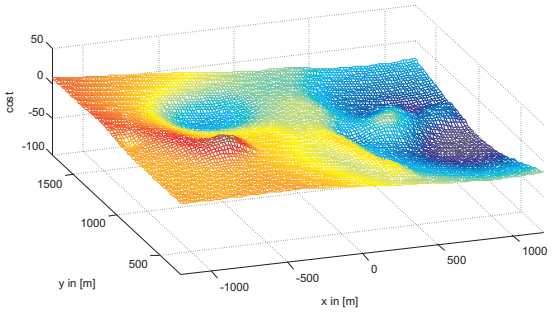
For moving objects the position and covariance for the various cost maps is predicted a fixed amount of seconds into the future. This improves the performance for tracking moving targets because agents will move to this future state. It is planned to change this static prediction to a dynamic prediction that depends on the travel time. For predicting a state, the same motion model (Section 3.3) is used as for all other application.

4.4. Mixed-Initiative Control

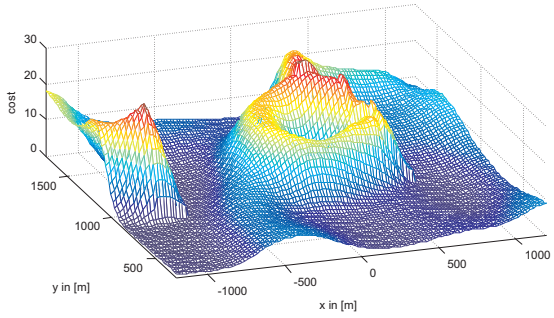
Each weight of the cost-functions can be changed to adapt to a new mission focus. This gives human operators the possibility to adjust the performance of the team. A graphical user interface has been created to allow easy bulk changes to these weights (Fig. 4.4).



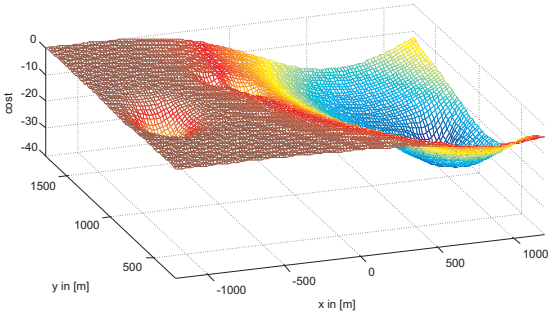
(a) Mission control space with region exploration map.



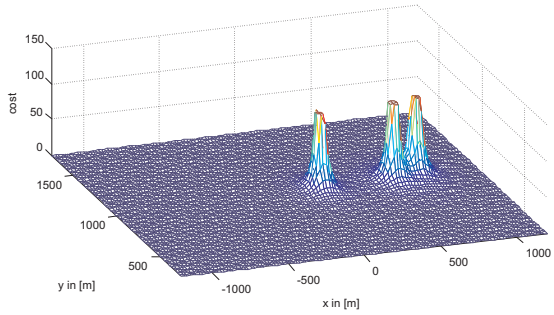
(b) Global mission achievement map.



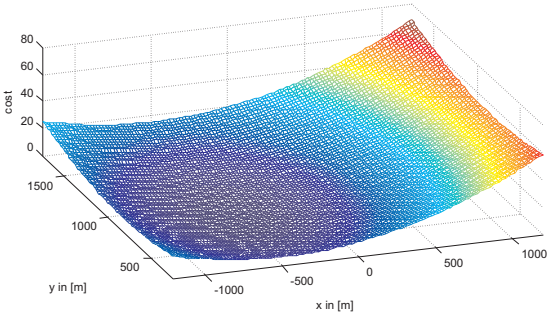
(c) Repulsive region exploration map.



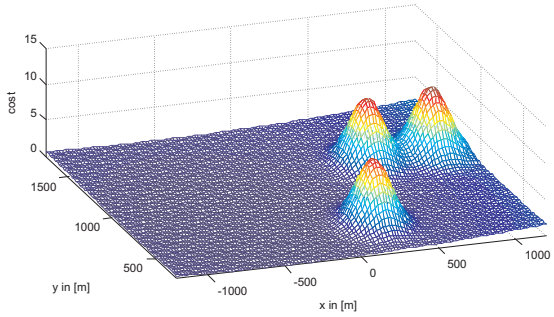
(d) Attractive target uncertainty map.



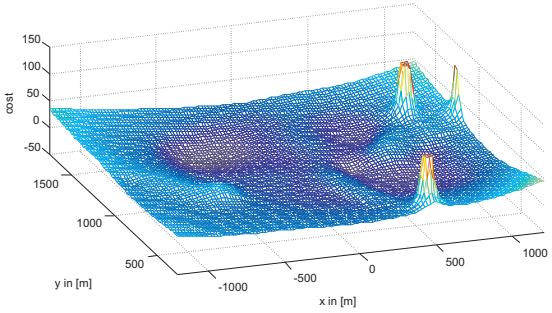
(e) Other agents repulsion map.



(f) Cost for travel time.



(g) Other agents' waypoints repulsion map.



(h) All summed and weighted cost maps.

Figure 4.3.: The controller is based on agent-specific cost maps whose minimum defines the next waypoint.

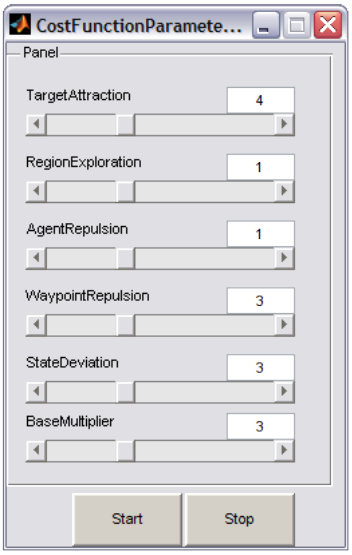


Figure 4.4.: The graphical user interface allows to make changes to the cost function weights during a running simulation.

5. Software Architecture

The proposed software architecture is an object-oriented simulation environment with a main focus on reusability and adaptability. This chapter describes our motivation for object-oriented software design and the structure of our software system.

The most common programming language for control system engineers is MatLab because of its potential in easy visualization, matrix handling, and comprehensive libraries with functions for algebra and control system. With the release of MatLab 2008a, object-oriented programming features are now included in the package. The combination of object-oriented benefits and the usability of the MatLab programming environment are a solid foundation for the development of the simulation environment.

5.1. Motivation for Object-Oriented Design

The driving force for designing this simulation environment was the need for a common ground in control algorithm development for the many researchers working on high level control at the C3UV. Up until now, each control algorithm ran purely on a simulation specifically designed for its own purpose. This makes combined simulations with different high level control algorithms nearly unfeasible since the code has to be rewritten in many aspects to comply with the other.

Code collaboration offers many opportunities to maintain and improve existing code and add specific control structures on a stable platform. To maintain the motivation in a team, the software architecture has to be easily understandable and expandable. This can be achieved with object oriented-programming.

5.1.1. Abstraction of Real World

Objects in OOP can be abstractions of objects in the real world which makes handling the objects highly intuitive. Looking at the object's properties and methods gives a general

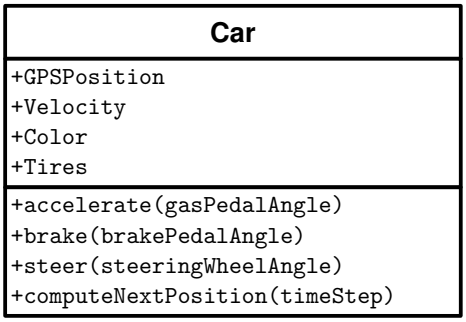


Figure 5.1.: Exemplary class interface of a car object that gives necessary information for understanding and implementation.

idea of what it can be used for. The abstract model of a car, as shown in Fig. 5.1, can be easily understood even by people without extensive programming knowledge.

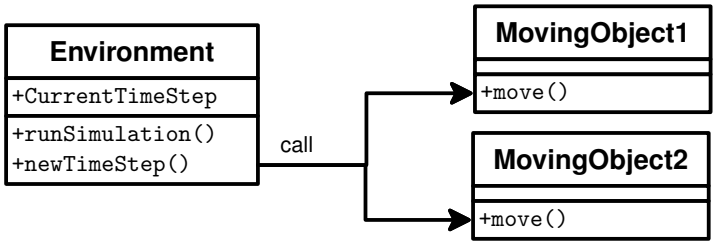
5.1.2. Reusability

In OOP, classes are a useful construct to encapsulate code and define interfaces over which the class interacts with its environment. The interface of a class is defined by its methods which define the tasks the class can perform and the class properties which define the state of the class (Fig. 5.1 for an example). Classes that have been properly designed for reusability can be reimplemented in a different environment with a basic understanding of what its public methods do instead of how they do it.

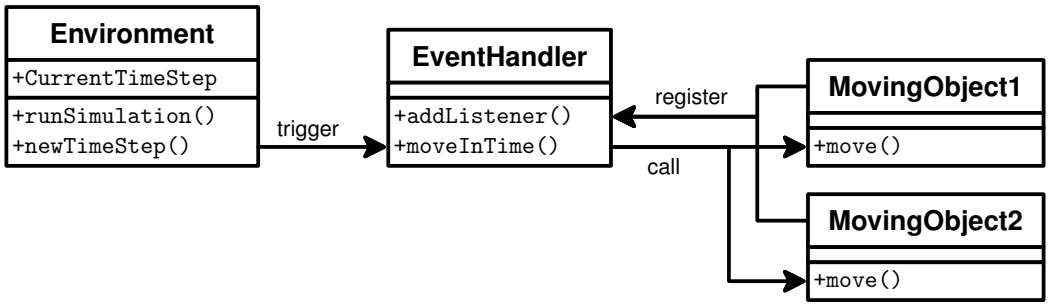
5.1.3. Inversion of Control

The inversion of control is a programming paradigm that reverses the way how function calls are executed. Also known as event-driven programming, it is often used in object-oriented programming. A caller function triggers an event instead of calling the other function directly [JF88]. This event is processed in a global event handler, which is included in event-based object-oriented programming languages. Other functions can subscribe themselves or be subscribed to by a specific event with a listener. Once the event is triggered, every function that has been registered to this event is executed in an arbitrary order.

Furthermore, inversion of control decouples the relationship between the caller function and the called function. The caller function does not (have to) know which function is listening. This simplifies the handling with different objects that are not specifically designed to communicate with each other.



(a) In sequential programming the caller function has to know and manage its related objects.



(b) Event-driven programming uses an event handler to decouple the triggering object from the effected objects.

Figure 5.2.: Inversion of control allows objects to register for specific events (Fig. 5.2(b)) instead of being called directly by another function (Fig. 5.2(a)).

This non-deterministic execution improves the workload especially on multi-core processors because the registered tasks can run in parallel. On the other hand, this parallel and non-deterministic process increases the the complexity and decreases the observability of the execution.

To guarantee a robust system, tasks have to run sequentially if they query the data which was manipulated in another task. This results in groups of parallel tasks that are triggered by events sequentially (similar to Fig. 5.4).

5.2. Structure

Our simulation environment is based on a hierarchical structure that represents the theoretical, mechanical and electrical objects of the real world.

Figure 5.3 illustrates the connection between the classes in the simulation. On the upper level, the class `Environment` is the global organization structure of the simulation. It handles time step management and contains all handles of objects in the simulation. Every object in the simulation knows the environment it belongs to (directly or as a child). Thus,

creating an object of the class `Environment` instantiates a basic set of simulation objects at the start. Other objects can be added or deleted dynamically before or during the simulation. Deleting this master object `Environment` results in the destruction of all the simulation objects.

The `Agent`, `Target`, and `TargetEstimator` classes are inherited by the `MovingObject` class. The `MovingObject` class defines the kinematic model that has been introduced in section 3.3. Each of these objects listens to the `moveInTime` event of the `Environment` object. Code examples of the `Environment.m`, `MovingObject.m` and `AgentController.m` can be found in the appendix at A.

The `Environment` object provides methods to add standard components to the system. To add an agent with one sensor, the command `agent = env.addAgent` is usually enough. If multiple regions are used (Fig. 5.5), the agent is normally positioned in the standard region. They can be created similarly: `region2 = env.addRegion` and to add a region estimator `restimate2 = env.addRegionEstimate`. For a region change, the `Region` property has to be changed: `agent.Region = region2`.

5.3. Version Control

A safe and stable work environment is essential for a smooth development in large projects. Version control is a major part in software development to ensure that developers can work at the same time on different parts of the same project without overwriting each other's changes. A history of code changes and contributions is saved with a unique version number and a comment about the committed changes on the version control server. This is used as a back up utility so that previous versions can be restored easily.

Furthermore, developers can create branches of the software where they can work on the software without affecting the main branch (generally the stable branch). After the instable development process has reached a final and robust state, this branch can be merged into the main branch to make the improved code accessible for everyone.

Subversion (SVN) is a version control system which is derived from and is mostly compatible with the Concurrent Versions System (CVS). However, it is in contrast to CVS still under active development and used as free software in many open source projects.

SVN stores all project data on a server from where clients can download or commit any version.

A client for SVN is necessary for accessing repositories. It has to run in the background to document changes, deletions and renames. TortoiseSVN is an SVN client for Windows

operating system which is integrated into the Windows Explorer interface. It gives the full functionality of SVN with an intuitive user interface.

5.4. Documentation

This simulation environment is designed to be attractive to other researchers so that they will work with it. Documentation can be seen as all means to help users understanding the program.

An easy and self-explanatory structure, as well as consistent and distinct naming, are forms of documentation. We designed our simulation framework in an object-oriented manner, so it is easy to see the different parts and the borderline in between objects. It gives a natural and intuitive understanding of how the software is composed. Furthermore, each object's properties explain themselves by their name. To change the maximum velocity of an agent, one would access the environment, choose an agent and change their maximum velocity: `environment.Agents1.MaxVelocity = [5;5]`. Each value is given in SI units.

In cases where the code gets more complicated, comments have been used to explain or summarize the behavior of functions. In general, long functions have been avoided by splitting them into several sub functions instead.

For further assistance, future directions and help about the repository, a help Wiki has been set up to facilitate a collaborative discussion ground and easily accessible information for adapting researchers.

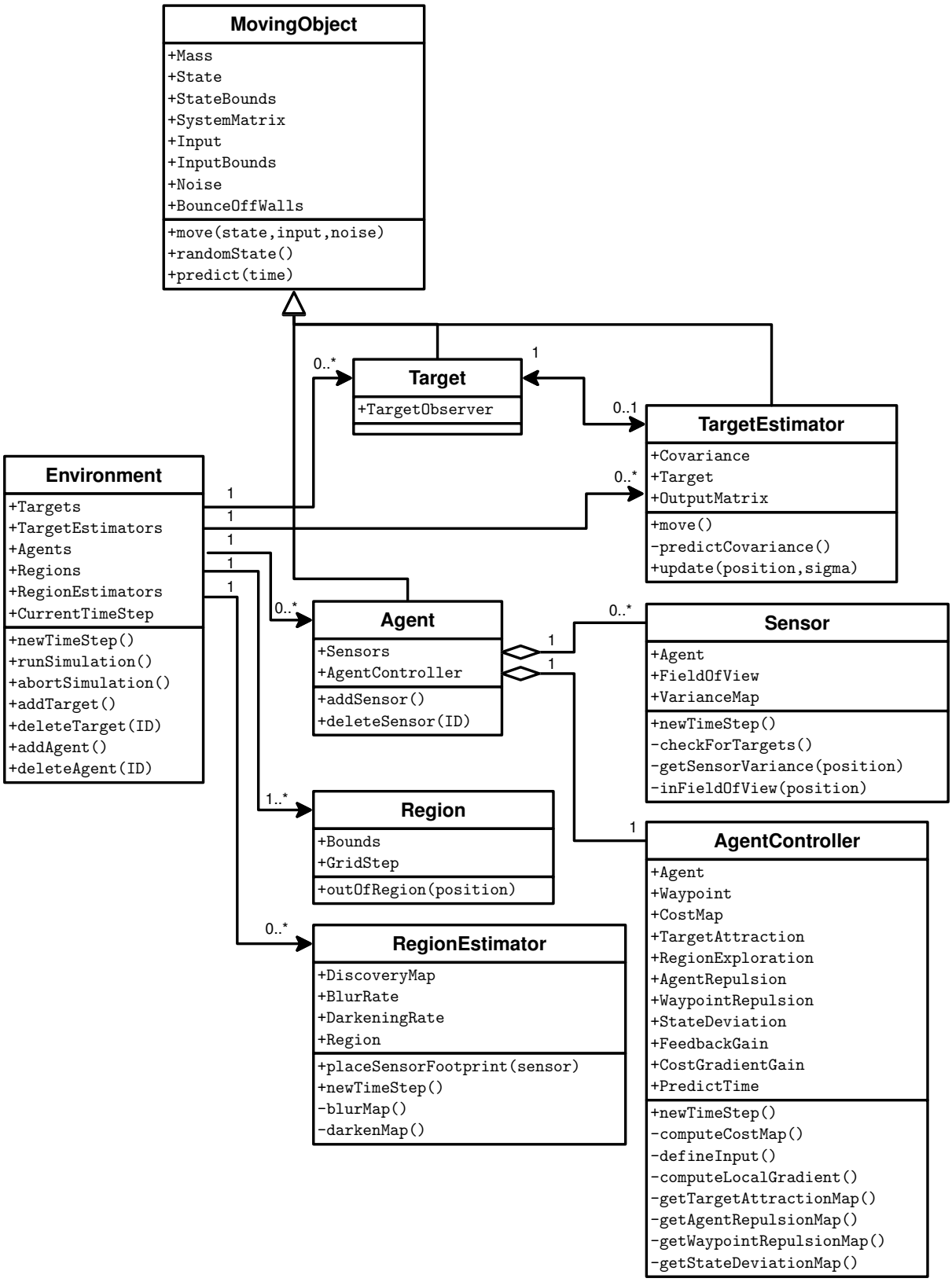


Figure 5.3.: This UML 2.0 class diagram visualizes the simplified network of objects in the simulation.

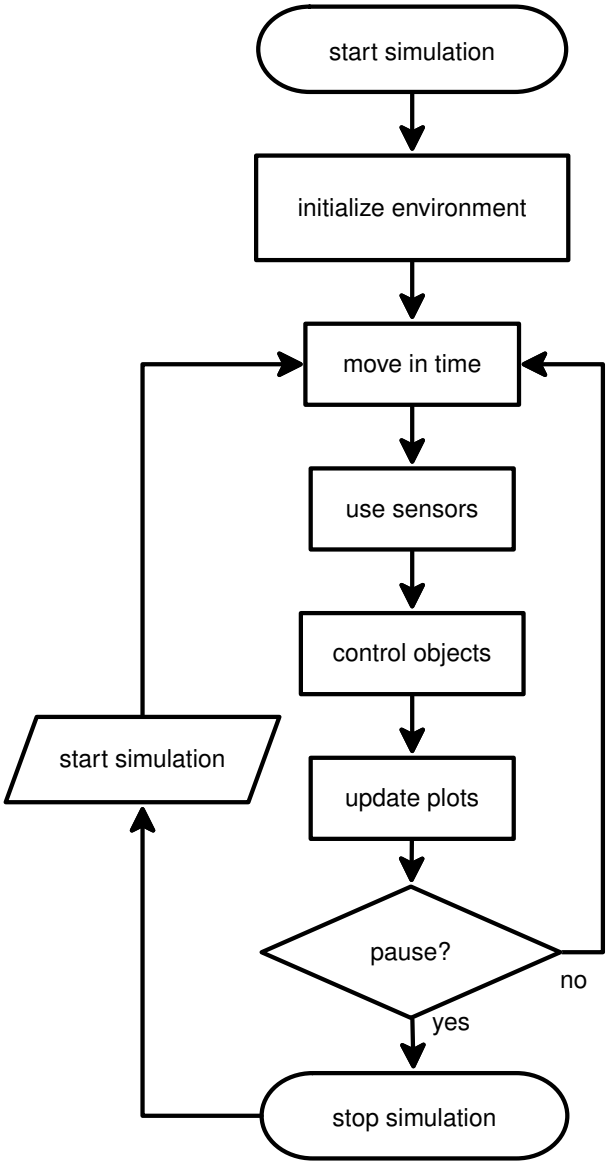
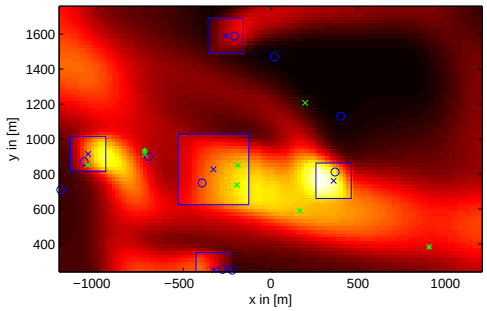
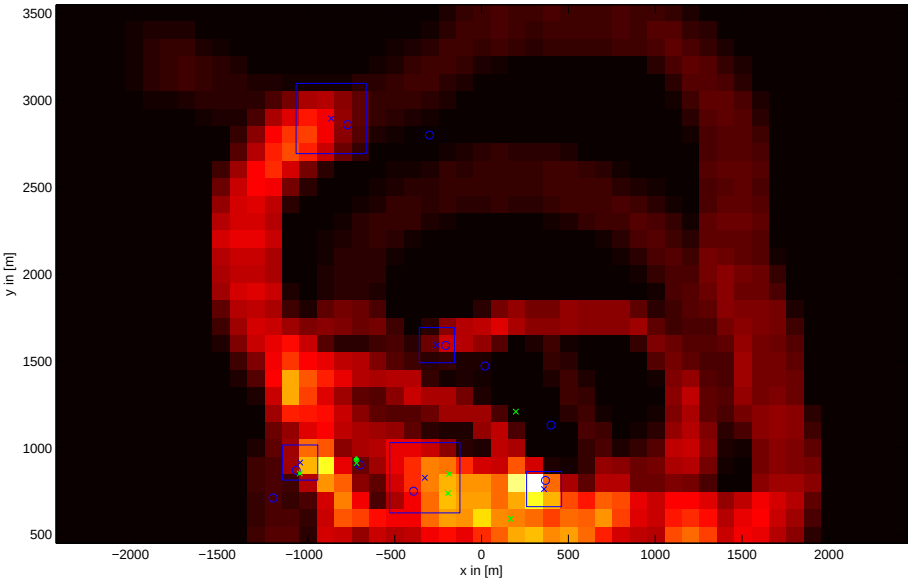


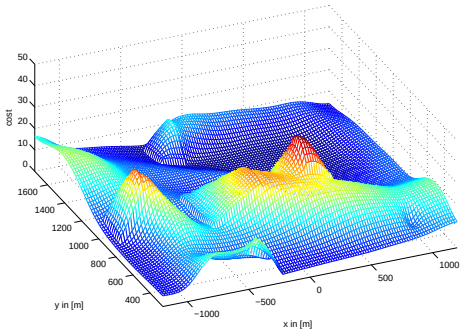
Figure 5.4.: The simulation runs a loop that triggers the main process steps



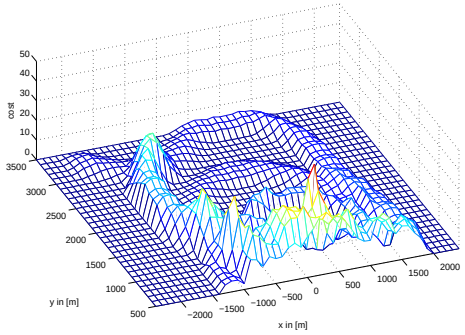
(a) A small high resolution region map.



(b) A large low resolution region map, overlapping the small one.



(c) Mission accomplishment for small map.



(d) Mission accomplishment for large map.

Figure 5.5.: Moving objects can operate on different regions and can change from one to another.

6. Conclusion and Outlook

During my research for this thesis, the focus of the project changed from designing a control system for multiple target recognition to designing a simulation framework for such controllers. With the growth of this framework the focus shifted back to the design of an innovative control algorithm to demonstrate the flexibility and functionality. The controller was meant to be a starting point, but turned out to perform very well within this simulation environment.

6.1. Summary of the Simulation Framework

This thesis has examined the challenges of collaborative control systems with the goal of finding an easy and intuitive approach to handle these problems. For the designing and testing of such controllers, a simulation framework has been created that is based on the modern principles of object-oriented programming. This framework consists of encapsulated classes that interact and react in an event-driven way to the progress of the simulation. This design results in great flexibility for collaborative software development and new contributions to the framework. Researchers adapting to this simulation environment benefit from the structural design and the distinct naming convention. This allows them to start with simulations with little effort and gain a deep understanding of the framework while working with it. Strictly handled encapsulation and inversion of control assure a robust platform that is easily extended and customized to the users' needs.

This framework allows users to dynamically add and delete multiple agents, targets, or regions and adapt to the changing mission parameters. Properties of objects are stored for each object individually and can be easily set for all objects in a batch. Graphical user interfaces allow parameter changes while running a simulation. For real-time mission analysis the graphical user interface listens to events triggered by changes of objects and are updated accordingly.

6.2. Summary of the Control Algorithm

The control algorithm is based on a virtual repulsive and attractive potential field approach. This concept has been improved and separated into two sub-problems. The first problem is to get our agent to move to the point of what we defined to be the minimum global cost. Consequently, all factors that affect the performance of the team of agents are summed up on a discretized region map. It follows that the choice of the waypoints depends on the travel time for the agent, other agents and their waypoints, the grade of exploration of the region, and the uncertainty of the targets. The waypoint following controller has a state feedback that assures a smooth convergence to our waypoint.

For compliance with our mission restrictions (e.g. avoiding no-fly-zones) the cost on the path in between agent and waypoint has to be analyzed. To achieve this, a local negative gradient feedback has been introduced which is added to the state feedback, thereby alternating the route to reduce the overall cost to reach the waypoint.

Since the latter is a local minimization approach and the aforementioned a global minimization without considering the cost along the way, we do not chose an optimal route. However, experiments show that this concept works reliably in our simulations and with a good performance easily up to 10 times faster than real-time.

The creation of the specific cost maps relies on many different parts. For the agent and waypoint cost maps we assume perfect communication between the agents. Similarly, for the region exploration map we assume to receive each sensors footprint. A model for the fading of the region exploration knowledge has been developed and implemented into the system. For the target uncertainty map, the targets' states are estimated by a Kalman filter which is updated on detection by a sensor. For now, perfect modeling is assumed throughout the simulation which means that the model of the estimated target is the same as the true target. Thus, only the measurement uncertainty has been considered.

6.3. Future Directions for the Simulation Framework

The simulation framework is a versatile starting ground that can be extended in many different directions. This depends on the future projects that it will be used for. Once the framework is used by different researchers, the road map on the Wiki will be the platform to discuss necessary improvements.

One important step is to include the possibility for rotation into the simulation. At the moment no object can be rotated. This change makes the simulation much more realistic

and problems occurring in the real system are more likely to be reproducible in the simulation. Rotation can be purely added to sensors, which is the most important step, but also to regions (to have multiple regions which are not aligned). Adding rotational states to the moving objects in our simulation will result in the nonlinear unicycle motion behavior. This change necessitates the target estimator to be improved from a linear Kalman Filter to an Extended Kalman Filter or particle filtering methods.

The assumption of perfect communication does not hold for the real world systems. Data loss and decreased bandwidth over distance needs to be considered in the model to account for the problems that occur during the flight tests. The agents act as a communication network that has to maintain connectivity at least through one link. Estimating the position of other agents provides a more robust communication and the possibility to return to a state with connectivity.

6.4. Future Directions for the Control Algorithm

The control algorithm is a first approach to show the flexibility and performance of the simulation environment. Thus, some features have been implemented with a heuristic approach where analytically proven methods are probable to perform better.

One problem that can occur (although never occurred during our simulations) with the mixed waypoint follower and gradient control is that an agent gets trapped in a U-shaped no-fly zone. A valid path that does not go over a certain cost limit can be found with maze theory. The ideal solution for this problem seems to be a recursive algorithm that goes from the agent to the waypoint in a direct way until it hits a wall. It would then split and travel in both directions along the wall until it can continue to go the direct way again.

An interesting and feasible improvement is to use continuous functions instead of grid based maps for our control structure. This allows for infinite regions, but global minimization becomes impossible. Where continuous functions are not an option, the particle filtering technique can be applied.

Up until now, this project assumes that a target can be identified and distinguished once it is found. For a real world use of this system this assumption does not hold in most cases, thus a data association filter has to be included. With similarity comparisons and evaluation of the probability of detection (from our Kalman Filter estimate), target association can be done with a high accuracy. The joint probability data association filter is a common approach for these problems.

Finally, a more systematic look-ahead function will probably increase performance. In our simulation, the controller estimates the state of moving objects in the simulation five seconds ahead. This can be formalized in the general control principle known as Model Predictive Control.

Bibliography

[BG08] E. Bitton and K. Goldberg. Hydra: A framework and algorithms for mixed-initiative UAV-assisted search and rescue. In *Automation Science and Engineering, 2008. CASE 2008. IEEE International Conference on*, pages 61–66, 2008.

[CGS08] David Cole, Ali Goktogan, and Salah Sukkarieh. *The Demonstration of a Cooperative Control Architecture for UAV Teams*, pages 501–510. 2008.

[CSG+06] David Cole, Salah Sukkarieh, Ali Goktogan, Hugh Stone, and Rhys Hardwick-Jones. *The Development of a Real-Time Modular Architecture for the Control of UAV Teams*, pages 465–476. 2006.

[FBLD06] T. Furukawa, F. Bourgault, B. Lavis, and H.F. Durrant-Whyte. Recursive bayesian search-and-tracking using coordinated uavs for lost targets. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 2521–2526, 2006.

[Hof08] Gabriel M. Hoffmann. *Autonomy for Sensor-Rich Vehicles: Interaction between Sensing and Control Actions*. PhD thesis, Dept. Aero. Astro., Stanford Univ., 2008.

[HWT06] G.M. Hoffmann, S.L. Waslander, and C.J. Tomlin. Mutual information methods with particle filters for mobile sensor network control. In *Decision and Control, 2006 45th IEEE Conference on*, pages 1019–1024, 2006.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2), 1988.

[KBD03] Zia Khan, T. Balch, and F. Dellaert. Efficient particle filter-based tracking of multiple interacting targets using an MRF-based motion model. In *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 1, pages 254–259 vol.1, 2003.

[Kuw07] Yoshiaki Kuwata. *Trajectory Planning for Unmanned Vehicles using Robust Receding Horizon Control.* PhD thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, February 2007.

[LFW08] Benjamin Lavis, Tomonari Furukawa, and Hugh F. Durrant Whyte. Dynamic space reconfiguration for bayesian search and tracking with moving targets. *Auton. Robots*, 24(4):387–399, 2008.

[NCT+05] B.Q. Nguyen, Yao-Ling Chuang, D. Tung, Chung Hsieh, Zhipu Jin, Ling Shi, D. Marthaler, A. Bertozzi, and R.M. Murray. Virtual attractive-repulsive potentials for cooperative control of second order dynamic vehicles on the caltech MVWT. In *American Control Conference, 2005. Proceedings of the 2005*, pages 1084–1089 vol. 2, 2005.

[RTG+07] A. Ryan, J. Tisdale, M. Godwin, D. Coatta, D. Nguyen, S. Spry, R. Sengupta, and J.K. Hedrick. Decentralized control of unmanned aerial vehicle collaborative sensing missions. In *American Control Conference, 2007. ACC '07*, pages 4672–4677, 2007.

[Rya08] Allison Ryan. *Information-Theoretic Control for Mobile Sensor Teams.* PhD thesis, University of California, Berkeley, 2008.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, (27):379–423 and 623–656, 1948.

[Tis08] John Patrick Tisdale. *Cooperative Sensing and Control with Unmanned Aerial Vehicles.* PhD thesis, University of California, Berkeley, 2008.

[TMD+06] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. Stanley: The robot that won the DARPA grand challenge: Research articles. *J. Robot. Syst.*, 23(9):692, 661, 2006.

A. Programming Examples

A small selection of programs are attached in this appendix for the reader to get an impression of the object-oriented MatLab simulation environment.

A.1. main.m

```
%   Created by Axel Hackbarth, UC Berkeley VDL / TU-Harburg MuM
%% init
addpath(genpath('helper'));

dbclif

close all
clear all
clear classes

dbstif  % comment for no error handling

%% simulation parameter definition
x = 10 % global scaling factor

EndTime = 20000          %#ok<NOPTS>  % in seconds
RegionSize = x*150*[1.6;1]  %#ok<NOPTS>  % in meters
RegionCenter = x*[0;100] %#ok<NOPTS>
RegionGridStep = x*2*[1;1]  %#ok<NOPTS>  % in meters / gridstep
TimeStepDuration = 1   %#ok<NOPTS>  % in seconds / timestep


nAgents = input('Enter num of agents: ')
nTargets= input('Enter num of targets: ')

%% simulation environment initialization
```

```
env = Environment();
this.Environment = env;
env.Region.Size = RegionSize;
env.Region.Center = RegionCenter;
env.Region.GridStep = RegionGridStep;
env.EndTime = EndTime;
env.TimeStepDuration = TimeStepDuration;


%% create agents with sensors
for i=1:nAgents
    agent = env.addAgent;
    agent.MaxVelocity = (i+2)*agent.MaxVelocity;
end


%% create targets
for i=1:nTargets
    target = env.addTarget;
    target.MaxVelocity = i/2*target.MaxVelocity;
end

%% create global cost function observer
c = GlobalCostFunction('Environment',env);

%% simulation start
CostFunctionParameterGUI;

notify(env,'startSimulation');
```

A.2. Environment.m

```
%   Created by Axel Hackbarth, UC Berkeley VDL / TU-Harburg MuM

classdef Environment < SimulationObject
    %Environment This class contains one region, targets and Agents
    %   Detailed explanation goes here

    properties
```

```
        Regions = {}
        CommHandler
        RegionEstimators ={}
        Agents = {}
        Targets = {}
        TargetEstimators = {}
        TargetEstimatorCoordinator

        CurrentTimeStep = 0
        SimulationIsRunning = false
        TimeStepDuration = 1 % seconds per timestep
        EndTime = 100        % seconds simulation time
        nextID = 1
        IDTable
        Visualize = true
    end
    properties (Dependent = true)
        Region
        RegionEstimator
        CurrentTime
    end

    events % for simulation progress
        moveInTime
        useSensors
        controlObjects
        updatePlots
        startSimulation
        stopSimulation
    end
    events % for simulation changes
        addedTarget
        deletedTarget
        addedAgent
        deletedAgent
        addedSensor
        deletedSensor
    end

    methods % set and get
```

```
        function time = get.CurrentTime(this)
            time = this.CurrentTimeStep * this.TimeStepDuration;
        end

        function region = get.Region(this)
            region = this.Regions{1};
        end
        function set.Region(this,region)
            this.Regions{1} = region;
        end
        function regionEstimator = get.RegionEstimator(this)
            regionEstimator = this.RegionEstimators{1};
        end
    end

    methods % initialization / change methods
        function this = Environment(varargin)
        %Environment constructor for simulation environment
        % Environment includes and creates all objects that are
           needed to
        % run a simulation.
        %
        % All class properties can be initialized by parameter /
           value
        % pairs individually:
        % this = Environment('propertyName',propertyValue)'
            this.ID = 0;

            % create and define region
            this.addRegion;

            optargin = size(varargin,2);
            if optargin > 1 && mod(optargin,2)==0
                for i=nargin-optargin+1 : 2 : optargin-1
                    this.(varargin{i}) = varargin{i+1};
                end
            end

            % create and define region Estimator
            this.addRegionEstimator;
```

```
        % create targetEstimatorcoodrinator
        this.TargetEstimatorCoordinator =
           TargetEstimatorCoordinator('Environment',this); %#ok
           <PROP,CPROP>

        % create listeners for start and stop simulation
        addlistener(this,'startSimulation',@(src,evdata)this.
           runSimulation);
        addlistener(this,'stopSimulation',@(src,evdata)this.
           abortSimulation);
    end % constructor

    function ID = getNewID(this,object)
        ID = this.nextID;
        this.nextID = this.nextID + 1;
        if exist('object','var')
            this.IDTable{ID} = object;
        end
    end

    function region = addRegion(this,varargin)
        region = Region('Environment',this,...
                                varargin{:}); %#ok<CPROP,PROP>
        this.Regions{end+1} = region;
    end

    function regionEstimator = addRegionEstimator(this,varargin
       )
        regionEstimator = RegionEstimator('Environment',this
           ,...
                                        'Region',this.Regions{
                                           length(this.
                                           RegionEstimators)
                                           +1},...
                                        varargin{:}); %#ok<CPROP,
                                           PROP>
        this.RegionEstimators{end+1} = regionEstimator;
    end
```

```
function agent = addAgent(this,varargin)
    agent = Agent('Environment',this,...
                  'LastTimeMoved',this.CurrentTimeStep
                     ,...
                  varargin{:});
    scale = agent.Region.ScaleFactor;
    agent.Noise = 0;
    agent.MaxVelocity = scale*[1;1];
    agent.InputBounds = scale*.5 * agent.InputBounds;
    agent.BounceOffWalls = false;
    % set sensor scaling factor
    agent.addSensor;
    for sensor=[agent.Sensors{:}]
        sensor.DiscretizationError = scale*sensor.
           DiscretizationError;
        sensor.FieldOfView = scale*sensor.FieldOfView;
    end
    this.Agents{end+1} = agent;

    notify(agent,'addedAgent')
    notify(this,'addedAgent',EventWithHandle(agent))

    %register plots
    figobjects = [this.RegionEstimators];
    for figobject = [figobjects{:}]
        if ~isempty(figobject.MapFigure)
            figure(figobject.MapFigure);
            agent.registerPlot;
            agent.AgentController.registerPlot;
        end
    end
end

% deletes agent by ID nr, otherwise last agent in list
function agent = deleteAgent(this,ID)
    for i = 1:length(this.Agents)
        if exist('ID','var') && this.Agents{i}.ID == ID
            break
        end
    end
```

```
            agent = this.Agents{i};
            this.Agents = this.Agents([1:i-1,i+1:end]);
            notify(this,'deletedAgent');
        end

        function target = addTarget(this,varargin)
            target = Target('Environment',this,...
                            'LastTimeMoved',this.CurrentTimeStep
                               ,...
                            varargin{:});
            scale = target.Region.ScaleFactor;

            dt = this.TimeStepDuration;
            target.Noise = scale*0.01*[0;0;1;1].*[dt^2/2;dt^2/2;dt;
               dt];
            target.MaxVelocity = scale*target.MaxVelocity;
            target.InputBounds = scale*target.InputBounds;
            target.randomState;

            this.Targets{end+1} = target;
            notify(this,'addedTarget',EventWithHandle(target));

            %register plots
            figobjects = [this.RegionEstimators];
            for figobject = [figobjects{:}]
                if ~isempty(figobject.MapFigure)
                    figure(figobject.MapFigure);
                    target.registerPlot;
                end
            end
        end
        % deletes target by ID nr, otherwise last target in list
        function target = deleteTarget(this,ID)
            for i = 1:length(this.Targets)
                if exist('ID','var') && this.Targets{i}.ID == ID
                    break
                end
            end
            target = this.Targets{i};
            this.Targets = this.Targets([1:i-1,i+1:end]);
```

```
            notify(this,'deletedTarget');
        end
    end

    methods % simulation methods
        function runSimulation(this, pauseTime)
            %runSimulation starts the simulation
            this.SimulationIsRunning = true;
            disp(['Started simulation at time step k=',int2str(this
                .CurrentTimeStep)]);
            disp(['Started simulation at simulation time t=',
                int2str(this.CurrentTime),' seconds']);
            tic
            while this.SimulationIsRunning && (this.CurrentTime <
                this.EndTime)
                this.CurrentTimeStep = this.CurrentTimeStep + 1;
                this.newTimeStep;
                if exist('pauseTime','var')
                    pause(pauseTime);
                end
            end
            disp(['Stopped simulation at time step k=',int2str(this
                .CurrentTimeStep)]);
            disp(['Stopped simulation at simulation time t=',
                int2str(this.CurrentTime),' seconds']);
            disp(['Simulation computational duration was ',num2str(
                toc),' seconds']);
        end

        function abortSimulation(this)
            this.SimulationIsRunning = false;
        end

        function newTimeStep(this)
        % every simulationobject has to have this function

            % ###### M O V E   O B J E C T S ############
            % this moves agents, targets and targetEstimators
            % regionEstimator is also processed
```

```
            % sensors might also include a move function and
               listener
            notify(this,'moveInTime');

            % ###### U S E   S E N S O R S ############
            notify(this,'useSensors');

            % ###### C O N T R O L   O B J E C T S ############
            notify(this,'controlObjects');

            % ###### U P D A T E   P L O T S ############
            if this.Visualize
                notify(this,'updatePlots');
            end
        end
    end
end
```

A.3. MovingObject.m

```
%   Created by Axel Hackbarth, UC Berkeley VDL / TU-Harburg MuM

classdef MovingObject < SimulationObject & Plottable
    %MOVINGOBJECT This class represents a moving object in its most
        general form
    %   Detailed explanation goes here

    properties
        LastTimeMoved = 0       % [s]
        Mass = 1                % [kg]
        SystemMatrix
        State                   % [m] and [m/s]
        StateArchive
        InputMatrix
        Noise                   % std deviation in [m] or [m/s]
        Input = [0;0]               % [m/s]
        StateBounds = [zeros(2);-1,1;-1,1] % [lower ; upper] %[m/s]
        InputBounds = 1*[-1,1;-1,1]    % [lower ; upper]  [m/s]
        BounceOffWalls = true
```

```
        StopAtWalls = true
        Region
    end % properties

    properties (Dependent = true)
        Position
        Velocity
        MaxVelocity
        NoiseCovariance
    end

    events
        changedState
        moved
    end


    methods
        function this = MovingObject(varargin) % constructor
            this = this@SimulationObject(varargin{:});
            if ~isempty(this.Environment)
                this.StateBounds(1:2,1:2) = this.Region.Bounds;
%                 this.StateBounds(3:4,1:2) = this.MaxVelocity;
                if isempty(this.State)
                    this.randomState;
                end
                dTime = this.Environment.TimeStepDuration;
                if isempty(this.SystemMatrix)
                    this.SystemMatrix = [eye(2),dTime*eye(2);zeros
                        (2),eye(2)];
                end
                if isempty(this.InputMatrix)
                    this.InputMatrix = [dTime^2/2*eye(2);eye(2)]./
                        this.Mass;
                    this.InputMatrix = [-dTime^2/2*eye(2);eye(2)]./
                        this.Mass;
                end
                % this makes the objects move on move event from
                % environment
                addlistener(this.Environment,'moveInTime',...
```

```
                @(src,evdata)this.move);
            addlistener(this.Environment,'updatePlots',...
                @(src,evdata)this.updatePlot);
        end
    end  % constructor

    function state = move(this, state, input, noise)
        if ~exist('state','var'), state = this.State; end
        if ~exist('input','var'), input = this.Input; end
        if ~exist('noise','var'), noise = this.Noise .* randn(
           size(this.State)); end
        state = this.SystemMatrix * state ...
              + this.InputMatrix  * input ...
              + noise;
        if nargout == 0
            this.StateArchive = this.State;
            this.State = state;
            notify(this,'moved');
        end
    end % move

    function state = randomState(this)
        statespan = this.StateBounds(:,2) - this.StateBounds
           (:,1);
        state = this.StateBounds(:,1) ...
              + statespan .* rand(size(statespan));
        if nargout == 0, this.State = state; end
    end % randomState

    function reverse(this)
        this.State(3:4) = - this.State(3:4);
    end % function

    function state = predict(this, time)
        if ~exist('time','var'), time = 1; end
        state = this.State;
        input = this.Input;
        for i=1:time/this.Environment.TimeStepDuration
            state = this.move(state,input);
            input = 0* input;
```

```
            end
        end

    end % methods

    methods % set and get methods
100     function set.State(this, state)
            % check for bouncing off at walls
            if this.BounceOffWalls || this.StopAtWalls
                if this.BounceOffWalls && (state(1) < this.
                   StateBounds(1,1)...
                                        || state(1) > this.
                                           StateBounds(1,2))
                    state(3) = -state(3);
                end
                if this.BounceOffWalls && (state(2) < this.
                   StateBounds(2,1)...
                                        || state(2) > this.
                                           StateBounds(2,2))
                    state(4) = -state(4);
110             end
                % check for stop at walls
                this.State(1:2,:) = max(this.StateBounds(1:2,1),...
                        min(this.StateBounds(1:2,2),state(1:2)));
            end

            % check for max velocity
            this.State(3:4,:) = max(this.StateBounds(3:4,1),...
                         min(this.StateBounds(3:4,2),state(3:4)));
            % check for normed velocity
120         factor = norm(this.State(3:4))/max(this.StateBounds
               (3:4,2));
            if factor > 1
                this.State(3:4,:) = this.State(3:4,:) ./ factor;
            end
            notify(this,'changedState');
        end % function

        function set.MaxVelocity(this,mvel)
            this.StateBounds(3:4,:) = mvel*[-1 1];
```

```
        end
        function mvel = get.MaxVelocity(this)
            mvel = this.StateBounds(3:4,2);
        end

        function set.Input(this,input)
            this.Input = max(this.InputBounds(:,1),...
                         min(this.InputBounds(:,2),input));
        end

        function pos = get.Position(this)
            pos = this.State(1:2);
        end
        function set.Position(this,pos)
            this.State(1:2) = pos;
        end

        function vel = get.Velocity(this)
            vel = this.State(3:4);
        end
        function       set.Velocity(this,vel)
            this.State(3:4) = vel;
        end

        function noisecovar = get.NoiseCovariance(this)
            noisecovar = diag( this.Noise.^2 .* ones(size(this.
               State)));
        end

        function region = get.Region(this)
            if isempty(this.Region)
                region = this.Environment.Region;
            else
                region = this.Region;
            end
        end
    end % set and get methods

    methods % plot methods
        function registerPlot(this)
```

```
            hold on
            x = this.Position;
            this.PlotHandle{end+1} = ...
                plot(x(1),x(2),this.PlotStyle);
            hold off
        end

        function updatePlot(this)
            x = this.Position;
            for i=1:length(this.PlotHandle)
                set(this.PlotHandle{i},'XData',x(1),'YData',x(2));
            end
        end
    end % methods

end % classdef
```

A.4. AgentController.m

```
%   Created by Axel Hackbarth, UC Berkeley VDL / TU-Harburg MuM

classdef AgentController < SimulationObject & Plottable
    %TARGETCONTROLLER Summary of this class goes here
    %   Detailed explanation goes here

    properties
        Agent
        Waypoint = [0;0]
        CostMap
        CostFigure
        CostGraphicsHandle
        BaseMap                     % is a region map that defines
           no-fly zones and zones to better avoid
        TargetAttraction = 4          % low: ignore / high: follow
           targets
        RegionExploration = 1       % low: ignore / high: consider
           map exploration
        AgentRepulsion = 1          % low: ignore / high: avoid
           agent collisions
```

```
        WaypointRepulsion = 3
        StateDeviation = 3   % low: global / high: local waypoint
           selection
        BaseMultiplier = 3   % low: global / high: local waypoint
           selection
        PredictTime = 5             % time horizon that is
           predicted for control
    end
    properties % for PD & gradient controller
        FeedbackGain = 0.4      % gain for defining sensor input by
            waypoint distance
        VelocityGain = 2        % gain for derivative feedback
        CostGradientGain = 2000  % gain for defining sensor input
           by local cost gradient
    end
    properties % for plotting
        PlotStyle = 'bo'
        Visualize  % leave empty for environment visualize setting
    end
    properties (Dependent = true)
        X
        Y
    end

    methods % set and get
        function x = get.X(this)
            x = this.Agent.Region.X;
        end
        function y = get.Y(this)
            y = this.Agent.Region.Y;
        end
        function vis = get.Visualize(this)
            if isempty(this.Visualize)
                vis = this.Environment.Visualize;
            else
                vis = this.Visualize;
            end
        end
    end % set and get
```

```
    methods (Access = public)
        function this = AgentController(varargin)
            this = this@SimulationObject(varargin{:});
            if ~isempty(this.Environment)
                this.BaseMap = this.createBaseMap;
                addlistener(this.Environment,'controlObjects',...
                    @(src,evdata)this.control);
                addlistener(this.Environment,'updatePlots',...
                    @(src,evdata)this.updatePlot);
            end
        end

        function control(this)
            this.computeCostMap;
            this.defineAgentInput;
        end
    end

    methods (Access = private)
        function baseMap = createBaseMap(this,X,Y)
            % is a region map that defines no-fly zones and zones
               to better
            % avoid
            if ~exist('X','var') || ~exist('Y','var')
                X = this.X;
                Y = this.Y;
            end
            center = this.Agent.Region.Center;
            baseMap = ((X-center(1)).^2 + (Y-center(2)).^2);
            baseMap = 1*baseMap / max(max(baseMap));
            if nargout == 0
                this.BaseMap = baseMap;
            end
        end

        %the input for the corresponding agent is generated
        function defineAgentInput(this)
            agent = this.Agent;
            % simple PD controller
            agent.Input = ...
```

```
                this.FeedbackGain * (this.Waypoint - agent.Position
                    ...
                    - this.VelocityGain * agent.Velocity ...
                    - this.CostGradientGain * this.
                       computeLocalGradient);
        end

        function computeCostMap(this,X,Y)
            if ~exist('X','var') || ~exist('Y','var')
                X = this.X;
                Y = this.Y;
            end

            DiscoveryMap = this.Agent.Region.RegionEstimator.
               DiscoveryMap;
%             DiscoveryMap = -this.Agent.Region.RegionEstimator.
   VarianceMap;

            Cost = this.BaseMultiplier      *this.BaseMap ...
                 + this.RegionExploration   *DiscoveryMap' ...
                 - this.TargetAttraction    *this.
                    getTargetAttractionMap(X,Y) ...
                 + this.AgentRepulsion      *this.
                    getAgentRepulsionMap(X,Y) ...
                 + this.WaypointRepulsion   *this.
                    getWaypointRepulsionMap(X,Y) ...
                 + this.StateDeviation      *this.
                    getStateDeviationMap(X,Y);

            % find minimum of cost map
            [minCost,idx1] = min(Cost);
            [minCost,idx2] = min(minCost);
            idx1=idx1(idx2);
            this.Waypoint = [X(idx1,idx2);Y(idx1,idx2)];
            this.CostMap = Cost;
        end

        function grad = computeLocalGradient(this)
            grad = [0;0];
            gradX = 0; gradY = 0;
```

```
        areaofinfluence = this.Agent.MaxVelocity...
                          *this.Environment.TimeStepDuration; %
                             meters in both directions
        agent = this.Agent;
        convert2grid = @(x)this.Agent.Region.
           convert2GridPosition(x);
        pos = agent.predict(this.PredictTime);
        pos = pos(1:2);
        if ~this.Agent.Region.outOfRegion(pos)
130     gridposbounds = [convert2grid(pos - areaofinfluence)
           ,...
                         convert2grid(pos + areaofinfluence)];

        cost = this.CostMap(gridposbounds(2,1):gridposbounds
           (2,2),...
                             gridposbounds(1,1):gridposbounds
                                (1,2));
        if ~any(size(cost)==[1 1])
            [gradX,gradY] = gradient(cost,...
                                 this.Agent.Region.GridStep(1)
                                    ,...
                                 this.Agent.Region.GridStep(2));
        end
140     grad = [mean(mean(gradX));
                mean(mean(gradY))];
        end
    end

    % calculate target attraction by gaussian maps
    function tMap = getTargetAttractionMap(this,X,Y)
        if ~exist('X','var') || ~exist('Y','var')
            X = this.X;
            Y = this.Y;
150     end
        tMap = 0;
        for i=1:length(this.Environment.TargetEstimators)
            targetobs = this.Environment.TargetEstimators{i};
            pos = targetobs.predict(this.PredictTime);
            sigma = 10*targetobs.Sigma;
            amp = sqrt(targetobs.Gain);
```

```
                tMap = tMap + gaussian2d(X,Y,pos(1),pos(2),sigma(1)
                    ,sigma(2),amp);
            end
        end

        % calculate agent repulsion by 1/dx maps
        function aMap = getAgentRepulsionMap(this,X,Y)
            if ~exist('X','var') || ~exist('Y','var')
                X = this.X;
                Y = this.Y;
            end
            aMap = 0;
            for i=1:length(this.Environment.Agents)
                agent = this.Environment.Agents{i};
                if agent.ID ~= this.Agent.ID
                    for j=1:length(agent.Sensors)
                        pos = agent.predict(this.PredictTime);
                        sigma = 0.5*(agent.Sensors{j}.FieldOfView
                            (:,2) - agent.Sensors{j}.FieldOfView
                            (:,1));
                        amp = 10;
                        % 1/x map
                        aMap = aMap + amp*min(10,(((pos(1) - X)/
                            sigma(1)).^2 + ((pos(2)- Y)/sigma(2))
                            .^2).^(-1));
                        % gaussian map
%                         aMap = aMap + gaussian2d(X,Y,pos(1),pos
    (2),sigma(1),sigma(2),amp);
                    end
                end
            end
        end

        % calculate planned agent's waypoint repulsion
        function wMap = getWaypointRepulsionMap(this,X,Y)
        % this part here introduces an order of execution - the
            first
        % who decides to move to a specific waypoint increases the
            cost
        % for all other agents who decide later
```

```
            if ~exist('X','var') || ~exist('Y','var')
190             X = this.X;
                Y = this.Y;
            end
            wMap = 0;
            for i=1:length(this.Environment.Agents)
                agent = this.Environment.Agents{i};
                if agent.ID ~= this.Agent.ID
                    for j=1:length(agent.Sensors)
                        pos = agent.AgentController.Waypoint;
                        sigma = 0.5*(agent.Sensors{j}.FieldOfView
                            (:,2) - agent.Sensors{j}.FieldOfView
                            (:,1));
200                     amp = 3;
                        % gaussian map
                        wMap = wMap + gaussian2d(X,Y,pos(1),pos(2),
                            sigma(1),sigma(2),amp);
                    end
                end
            end
        end

        % get agent state deviation cost
        function dMap = getStateDeviationMap(this,X,Y)
210         % the further away the
            % waypoint from the predicted next timestep position,
               the
            % higher the cost to get there
            % this actually is a decision between local and global
               waypoint
            % finding
            if ~exist('X','var') || ~exist('Y','var')
                X = this.X;
                Y = this.Y;
            end
            state = this.Agent.predict(this.PredictTime);
220         maxvel = this.Agent.MaxVelocity;
            dMap = 1e-2*((X - state(1)).^2 + (Y - state(2)).^2)...
                    / (maxvel(1)^2 +maxvel(2)^2) ;
        end
```

```
    end

    methods (Access = public) % plot functions
        function registerPlot(this)
            x = [10*this.Agent.Input + this.Agent.Position,this.
               Waypoint];
            hold on
            this.PlotHandle{end+1} = ...
                plot(x(1,:),x(2,:),this.PlotStyle);
            hold off
        end

        function updatePlot(this)
            % P L O T $$$$$$
            if this.Agent.ID == this.Environment.Agents{1}.ID
                this.showCostMap;
            end

            x = [10*this.Agent.Input + this.Agent.Position,this.
               Waypoint];
            for i=1:length(this.PlotHandle)
                set(this.PlotHandle{i},'XData',x(1,:),'YData',x
                   (2,:));
            end
        end

        function showCostMap(this)
            if this.Visualize
            if isempty(this.CostFigure)
                this.CostFigure = figure;
                set(gcf,'Name',['AgentController ',num2str(this.ID)
                   ,' Cost Map']);
                setFigurePosition(2,2,4)
%                 set(gcf,'WindowStyle','docked')
                this.CostGraphicsHandle = mesh(this.Agent.Region.X
                   ,...   % for 3D-plot
                                                this.Agent.Region.
                                                   Y,...
                                                this.CostMap);
                xlabel('x in [m]');
```

```
                ylabel('y in [m]');
                zlabel('cost');
%                 view(0,90)
%                 axis square
                axis tight
                zlim('auto')

            else
                set(this.CostGraphicsHandle,'ZData',this.CostMap);
                    % for 3D-plot
            end
            end
        end
    end
end
```