Distributed Spatial Predictive Formation Control - Laboratory development and experimental study

J.F. de Winkel

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Distributed Spatial Predictive Formation Control - Laboratory development and experimental study

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

J.F. de Winkel

March 26, 2017

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Abstract

This thesis describes the development of the Delft Center for Systems and Control (DCSC) Networked Embedded Robotics Lab (NER) and a novel distributed formation control algorithm to showcase the lab's experimental capabilities. The lab environment is built to support ground as well as aerial robotic platforms featuring a netted environment and state-of-the-art camera localization system. Both the hardware and software infrastructure considered in this work were chosen to provide an easy interface for students to run distributed experiments with a focus on code reusability. To this end, the Robot Operating System (ROS) was chosen as a software framework as it provides drivers for most robotics platforms, libraries for common tasks such as localization and navigation, communication between software nodes on different hardware, and its widespread use and acceptance by the robotics community. The developed algorithm to showcase the multi-robot coordination capabilities of the laboratory uses a model predictive based formation controller with a coordinate transform from Euclidean to spatial coordinates where the agents' position is expressed in terms of traveled path length and path deviation. The method was implemented on multiple iRobot Create platforms with independent computing power. A distributed formation controller was achieved by applying a consensus algorithm on the traveled path length. Results show that the desired formations can be achieved even while tracking a target path. Implementation of the complete experiment in ROS illustrates that the developed laboratory setup and its components (hardware and software architecture) are capable of running distributed robotics experiments.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to sincerely thank all who assisted me during my work on this thesis, most notably my supervisor dr.ir. Tamás Keviczky for his patience and support. I hope my work on the robotics enviroment for DCSC will be of continued use in the future.

Delft, University of Technology
March 26, 2017

J.F. de Winkel

Chapter 1

Introduction

1-1 Motivation

Research on robotics platforms often requires extensive knowledge in multiple domains, such as electrical engineering, mechanical engineering, and computer science. For control students, however, the emphasis is placed on developing novel control algorithms. Practical implementation of these algorithms requires work in all aforementioned domains. To provide as much help as possible in this practical implementation a reliable and functional lab environment is necessary. Such a lab environment means providing students with a plug-and-play hardware and software infrastructure, detailed manuals and a collection of experiments allowing them to choose and run existing algorithms and only modify parts relevant to their research. To provide such a lab environment a highly modular software framework will be required with support for a broad set of hardware components.

The primary research goal of the lab environment to be developed are distributed, or networked, experiments and as such this thesis highlights the development and practical implementation of a novel distributed spatial predictive formation control method showcasing the capabilities of the new DCSC Networked Embedded Robotics Laboratory. This method requires localization, state estimation, consensus and control of multiple robotic agents and will, therefore, provide examples on all relevant topics to control students working on a practical implementation of their experiment in the Networked Embedded Robotics Lab.

Spatial Predictive Control (SPC) is a relatively new approach where coordinates of an agent are expressed not in Euclidean space but in length traveled along a given path s and a tangential offset from this path e_y. This method is highly suitable for path following and obstacle avoidance and can be used as a high-level path planning method, for instance for automated car-like robots. Extending this method to support multiple agents to move as a formation along a given path provides an interesting research topic and can easily be extended to different platforms.

1-2 Research Objectives

This thesis work distinguishes between the following research objectives:

- Develop a laboratory environment consisting of multiple robotic platforms, computers to use for control algorithms and development, a motion capture system to track robots and the necessary hardware to support communication between all components.
- Provide software support for multiple, ground as well as aerial, robotics platforms.
- Focus on usability and reusability of the setup for future students by providing detailed instructions on working with hardware and reusable code.
- Implement multiple demos showcasing the lab's platforms and capabilities.
- Develop a novel distributed formation control approach for path following based on Spatial Predictive Control and consensus algorithms.

1-3 Thesis organization

In the following chapters, the development of the distributed formation control algorithm, the laboratory setup, and its components, and the obtained experimental results will be discussed. Chapter 2 provides the mathematical background on the Spatial Predictive Control and Consensus algorithms and the developed formation control algorithm that this thesis work contributes. In Chapter 3, details are given on available software infrastructures for robotics and the choices made in engineering the laboratory setup with a detailed view of all components and robotic platforms that are available to students. Chapter 4 provides the results of the practical implementation of the formation control experiment in the laboratory setup. Finally, Chapter 5 contains conclusions on the overall result of the developed laboratory setup and experiment as well as recommendations for future improvements.

Chapter 2

Distributed Spatial Predictive Formation Control

The research done in this thesis consists of a combination of formation control and a state-of-the-art path tracking method called Spatial Predictive Control. This combination is researched with the aim of developing a distributed formation controller for a group of unicycle type robots. First, an overview of formation control methods is given with a focus on consensus approaches, which will be used to describe information flow between agents in a formation. Then the mathematical background is provided for the Spatial Predictive Control method that is used to provide path tracking capabilities to the mobile agents. These two methods are then combined for agents with unicycle dynamics. Simulation results are provided for the resulting control algorithm.

2-1 Formation control

Formation control is an important subject within the study of multi-agent systems as it has the possibility to increase the overall system effectiveness compared to single agent systems. Some examples are the cooperative movement of large objects, exploration or search and rescue missions. Also, the option of using cheaper hardware and the lower chance of failure compared to using a single agent are important reasons for using multi-agent systems. Formation control for these multi-agent systems is an extensively studied subject with multiple solutions. Typically these methods are categorized as leader-follower, virtual structure, potential field, and consensus methods [9].

Leader-follower methods work by appointing one mobile agent as a leader whilst another agent is appointed as a follower. Control laws for follower agents are designed based on a distance keeping constraint between the leader and the follower. Two types of distance constraints are most prominent, l-l and l-ϕ controllers. In l-l control, Figure 2-1a, one agent keeps a fixed distance to two leader agents. In l-ϕ control, Figure 2-1b, the follower keeps a relative distance and angle to a single leader agent. The leader-follower approach can be implemented using

any control structure, for instance using feedback linearization [10] or optimization algorithms [11].

(a) Relative distance control

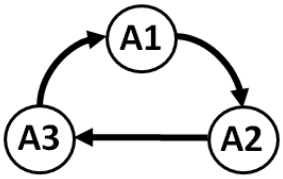
(b) Relative distance and angle control

As an extension of leader-follower control, virtual structures were introduced. In a virtual structure algorithm, instead of following an existing leader agent, the leader is defined in software. This adaptation makes a formation control algorithm more robust as the failure of a leader agent does not comprise it [12]. Typically in virtual structure approaches the leader is defined as a rigid body containing the locations of all agents in the formation. From the position or trajectory of this virtual rigid body, the desired position and trajectories for other agents are computed using their desired offset from the rigid bodies' center of mass.

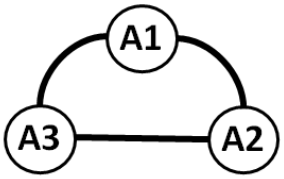
Potential field methods work by creating artificial potentials between an agent and its goal. These potentials can be viewed on a two-dimensional plane as peaks and valleys. The gradients on these potential fields are used as force or velocity inputs for a mobile agent such that it's attracted to valleys and pushed away from peaks. This method is typically used for path generation purposes, such as [13], but is also applicable to formation control [14]. The main issue with potential field approaches is the existence of local minima within the field, allowing an agent to get stuck. Workarounds for this problem exist, such as adding a centrifugal force to the input force computed from the potential field [15].

Consensus methods for formation control focus on using a description of information flow between multiple agents to design control laws that prove that a formation, given a certain connection topology, converges. The parameters that are used for convergence are referred to as the information state and the connection topology of a formation is described using graph theory [16]. Graph theory describes connectivity using graphs that consist of nodes $\mathcal{N}$, representing agents within a formation, and edges $\mathcal{E}$, representing an information sharing connection to another agent. Graphs can either be directed (information flows along an edge in only one direction) or undirected (information flow is bi-directional) and they can be connected (a path from one node to another is always available) or disconnected (there are one or multiple nodes that cannot be reached from another node) [16]. Figure 2-2a shows an example of directed graph, where agent communication is one-directional. Figure 2-2b shows an example of an undirected graph where communication is bi-directional. Figure 2-2c shows an example of a graph without a spanning tree, meaning that not all nodes in the graph receive information from others. The previously mentioned leader-follower and virtual structure methods can also be described as a consensus problem where information flows in only one way, a directed graph, from the (virtual) leader to the follower.

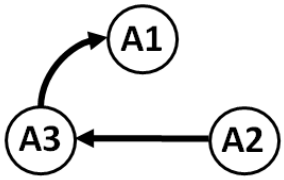
In [17], graph based consensus approaches are suggested for formation keeping whilst following a constant or time-varying reference, a trajectory, for systems with dynamics $\dot{\xi} = u_i$. A



(a) Directed graph with spanning tree



(b) Undirected graph with spanning tree



(c) Directed graph without spanning tree

formation keeping consensus algorithm is given for connected, directed or undirected, graphs for situations where the reference is or is not known to all agents. For these consensus methods, a graph $\mathcal{G}$ described by the pair $(\mathcal{N}, \mathcal{E})$ is used with an adjacency matrix A, a square matrix that indicates whether nodes are connected not. It is given by $A = [a_{ij}]$ where $a_{ii} = 0$ and $a_{ij} > 0$ if $(j, i) \in \mathcal{E}$. In the case of multi-agent systems that should converge to a constant reference state, the control input for each agent is given as (2-1).

$$u_i = -\sum_{j=1}^{n} g_{ij} k_{ij} (\xi_i - \xi_j) - g_{i(n+1)} \alpha_i (\xi_i - \xi^r) \tag{2-1}$$

Where $k_{ij} > 0$ and $\alpha = 0$ are a weights and $g_{ij} = 1$ if information flows from agent i to j as given in the adjacency matrix and otherwise $g_{ij} = 0$. Similarly if the agent i) has access to the reference state then $g_{i(n+1)} = 1$ and otherwise $g_{i(n+1)} = 0$. As can be seen, the velocity input for this system becomes a weighted sum of formation errors and path tracking errors. When following a trajectory the reference state is time varying, $\dot{\xi^r} = f(t, \xi^r)$. In this case, if the path is known to all agents, Figure 2-3a, (2-2) guarantees convergence for the formation along the path.

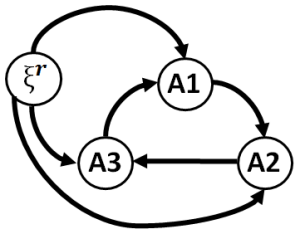
$$u_i = g_{i(n+1)} f(t, \xi^r) - \sum_{j=1}^{n} g_{ij} k_{ij} (\xi_i - \xi_j) - g_{i(n+1)} \alpha_i (\xi_i - \xi^r) \tag{2-2}$$

If the reference state however is time varying and not known to all agents, Figure 2-3b, (2-3) is used.

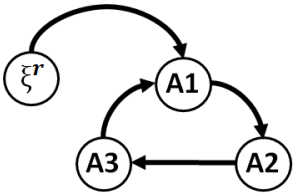
$$\begin{aligned} u_i = \frac{1}{\eta_i} \sum_{j=1}^{n} g_{ij} k_{ij} [\dot{\xi}_j - \gamma_i (\xi_i - \xi_j)] + \frac{1}{\eta_i} g_{i(n+1)} \alpha_i [f(t, \xi^r) - \gamma_i (\xi_i - \xi^r)] \\ \eta_i = g_{i(n+1)} \alpha_i + \sum_{j=1}^{n} g_{ij} k_{ij} \end{aligned} \tag{2-3}$$

As can be seen, depending on the communication topology and availability of the reference information state for all agents, solutions exist to reach convergence of an information state.

In this section, four formation control methods have been described. All of these methods can be used for centralized or distributed control. The consensus approach will be implemented as it is the most general method allowing the study of multiple different connection topologies.



(a) Reference path available to all agents



(b) Reference path available to single agent

2-2 Spatial Predictive Control

Spatial Predictive Control is a type of Model Predictive Control used for path tracking purposes, where the dynamics of a vehicle are expressed in a spatial-dependent form rather then time-dependent. This section provides a short mathematical background on Model Predictive Control for trajectory tracking, using a unicycle type robot, after which the transformation to spatial-dependent dynamics is provided.

2-2-1 Trajectory tracking using Model Predictive Control

Differential drive, or unicycle, type robots, Figure 2-4, are simple robots with two separately actuated wheels and one free wheel. This type of robot has nonholonomic and nonlinear dynamics as given in (2-4).

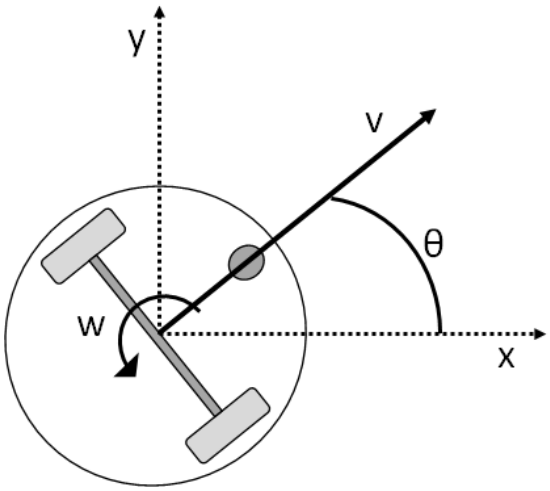


Figure 2-4: Unicycle robot

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos(\theta) \\ v\sin(\theta) \\ w \end{bmatrix} \tag{2-4}$$

The error dynamics given a desired reference pose can be described as (2-5).

$$\begin{bmatrix} \dot{x_e} \\ \dot{y_e} \\ \dot{\theta_e} \end{bmatrix} = \begin{bmatrix} \cos(\theta_r) & \sin(\theta_r) & 0 \\ -\sin(\theta_r) & \cos(\theta_r) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - x_r \\ y - y_r \\ \theta - \theta_r \end{bmatrix} \tag{2-5}$$

When tracking a trajectory, the reference pose $\begin{bmatrix} x_r & y_r & \theta_r \end{bmatrix}^T$ will be time-varying. Model Predictive Control (MPC) is a technique that can be used to predict the tracking errors, as defined by the error dynamics, over a finite horizon and find control inputs that minimize this error. Each sampling time, the predicted tracking error is defined as an optimization problem. This optimization problem is then solved after which the first optimal input in the calculated sequence of inputs is applied to the system. In the next sampling time, this process is repeated. For any reference trajectory the general optimal control problem at time t is as (2-14) [18].

$$\min_u \int_t^{t+T_p} F(x(\tau), u(\tau)) d\tau \tag{2-6}$$

subject to:

$$\begin{aligned} \dot{x}(\tau) &= f(x(\tau), u(\tau)) \\ u(\tau) &\in \mathcal{U} \qquad \forall \tau \in [t, t + T_c] \\ x(\tau) &\in \mathcal{X} \qquad \forall \tau \in [t, t + T_p] \\ \|x(t + T_p)\|_P &\leq \beta \|x(t)\|_P \qquad \beta \in [0, 1) \end{aligned} \tag{2-7}$$

With $F(x, u) = \|x\|_Q + \|u\|_R$ and Q and R are positive definite matrices used as weights on the tracking error and input respectively. T_p represents the length of the prediction horizon, T_c represents the control horizon with $T_c \leq T_p$. Finally (2-7) provides input constraints $\mathcal{U}$, state constrains $\mathcal{X}$, and a contractive constraint that defines the required amount of contraction between the initial and final state.

2-2-2 Conversion to spatial dynamics

In the optimal control problem (2-14) a trajectory tracking controller was designed. A trajectory is a path parametrized in time, e.g. a set of poses $p(t) = \begin{bmatrix} x(t) & y(t) & \theta(t) \end{bmatrix}^T$ that describe where an agent should be at each moment in time. This assumes that a trajectory generation algorithm is used that is capable of generating feasible poses in time without violating vehicle dynamics constraints. A more general approach would be using a path, a set of poses that is not parametrized in time, and define the optimal control problem such that it tracks the path in a desired amount of time regarding the vehicle dynamics and constraints. To this end, the optimal control problem can be reformulated in such a way that it can be solved in a time-invariant manner. In [1, 19] this problem is solved by applying a transformation from time-dependent agent dynamics to spatial-dependent dynamics. These spatial-dependent dynamics require expressing the pose of a mobile agent in terms of its progression s along a known reference path and its deviation in distance and heading angle e_y and e_ψ respectively. This transformation is typically done using a Frenet frame [20]. A Frenet frame is a frame that can be defined on each point of the differential geometry of a curve. It is defined by

a vector tangent $\overrightarrow{\hat{T}}(s)$, and vector normal $\overrightarrow{\hat{N}}(s)$ to a curve as shown in Figure 2-5. These vectors are defined as (2-8)[19].

$$\begin{aligned} \overrightarrow{T}(s) &= \frac{\partial^2 f_{path}(s)}{\partial s}, \qquad & \overrightarrow{\hat{T}}(s) &= \frac{\overrightarrow{T}(s)}{\|\overrightarrow{T}(s)\|_2} \\ \overrightarrow{N}(s) &= \frac{\partial^2 f_{path}(s)}{\partial s^2}, \qquad & \overrightarrow{\hat{N}}(s) &= \frac{\overrightarrow{N}(s)}{\|\overrightarrow{N}(s)\|_2} \end{aligned} \tag{2-8}$$

The desired heading at each point on the path is defined by the angle between tangent vector and the x-axis whilst the curvature of the path is defined by magnitude of the normal vector.

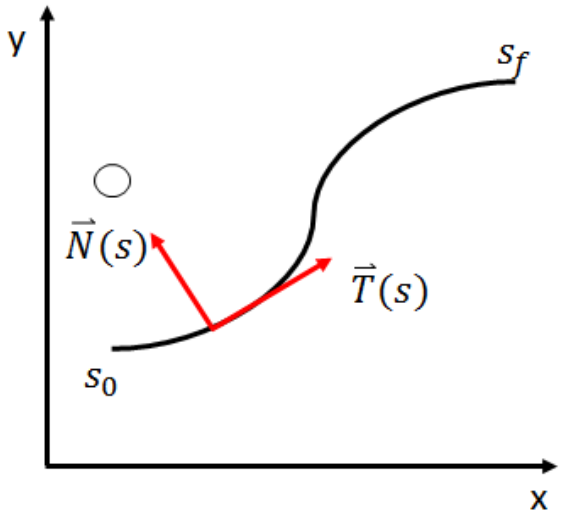


Figure 2-5: Construction of a Frenet frame on a curved geometry

Using the above transformation of a Cartesian coordinate frame to a Frenet frame the dynamics of the mobile agent can be transformed as depicted in Figure 2-6. The state vector for the mobile agent is defined as $\xi^s = \begin{bmatrix} \dot{y} & \dot{x} & \dot{\psi} & e_\psi & e_y \end{bmatrix}^T$ where $\dot{x}$ and $\dot{y}$ represent vehicle velocity along the corresponding axis, ψ is the path heading angle and $\dot{\psi}$ its rate of change, and e_ψ and e_y represent the vehicle heading angle and position error with respect to the path respectively.

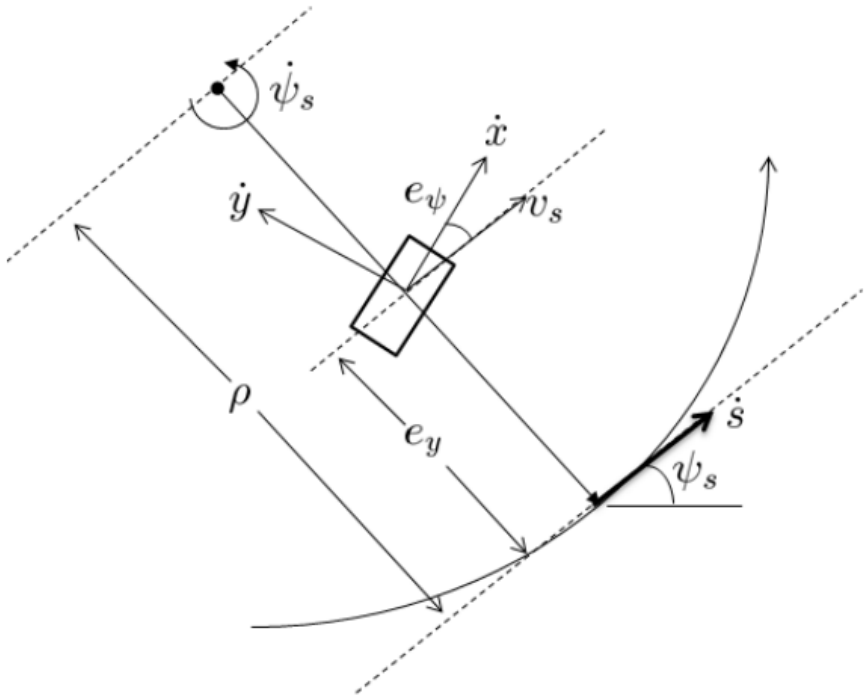


Figure 2-6: Definition of spatial-dependent dynamics [1]

The vehicle velocity along the path $v(s)$ can be defined in terms of angular velocity and radius of curvature. By projecting the absolute vehicle velocity in $\dot{x}$ and $\dot{y}$ in the direction of the reference heading, a transformation from velocity defined in the Cartesian coordinate frame to the Frenet frame can be made.

$$\begin{aligned} v_s &= (\rho - e_y) \cdot \dot{\psi}_s \\ &= \dot{x} \cdot \cos(e_\psi) - \dot{y} \cdot \cos(e_\psi) \end{aligned} \tag{2-9}$$

Where v_s is the speed of the mobile agent along the given path, ρ is the radius of curvature and ψ_s is vehicle heading at the current position along the path.

$$\dot{s} = \rho \cdot \dot{\psi}_s = \frac{\rho}{\rho - e_y} \cdot (\dot{x} \cdot \cos(e_\psi) - \dot{y} \cdot \cos(e_\psi)) \tag{2-10}$$

Using the following simple relationships it is possible to express the path and heading error of the vehicle as derivatives of the traveled path length s.

$$\xi^{'} = \frac{d\xi}{dt}\frac{dt}{ds} = \frac{\dot{\xi}}{\dot{s}} \tag{2-11}$$

$$e_\psi^{'} = (\psi - \psi_s)^{'} = \frac{\dot{\psi}}{\dot{s}} - \psi_s^{'} \tag{2-12}$$

$$e_y^{'} = \frac{\dot{e}_y}{\dot{s}} = \frac{\dot{x} \cdot \cos(e_\psi) - \dot{y} \cdot \cos(e_\psi)}{\dot{s}} \tag{2-13}$$

Where it should be noted that the time as a function of s can be retrieved by integrating $t^{'} = \frac{1}{\dot{s}}$. The resulting error dynamics can be used to define a new optimal control problem

with respect to s in stead of t that aims at minimizing tracking errors and maintaining a reference velocity along the path.

$$\min_u \int_t^{s+H_p} F(\eta(s), \eta_{\text{ref}}(s), u(s), \frac{du}{ds}(s)))ds \tag{2-14}$$

subject to:

$$\frac{d\xi}{ds}(s) = f(\xi(s), u(s)) \tag{2-15}$$

Where $\eta = \begin{bmatrix} \dot{x} & \dot{\psi} & e_\psi & e_y \end{bmatrix}^T$ and $F(\eta, \eta_{\text{ref}}, u, \frac{du}{ds}) = \|\eta - \eta_{\text{ref}}\|_Q + \|u\|_R + \|\frac{du}{ds}\|_S$ with Q, R and S positive definite matrices as weights on tracking error, input, and input change respectively. The weight on input change is added to generated smoother inputs. Discretizing the error dynamics using $\frac{d\xi}{ds}(s) = \frac{\xi(s+\Delta s) - \xi(s)}{\Delta s}$ allows the optimal control problem to be formulated as (2-16).

$$\min_u \sum_{k=s}^{s+H_p} \|\eta_{k,s} - \eta_{ref_{k,s}}\|_Q^2 + \|u_{k,s}\|_R^2 + \|\Delta u_{k,s}\|_S^2 \tag{2-16}$$

The resulting controller calculates the optimal linear and angular input velocities that allow an agent to track a reference path with desired velocity. Note that the state η contains terms for path deviation and angle error. By setting these terms to zero, an agent will follow the reference path exactly. Using $e_y \neq 0$ will generate control inputs such that an agent will follow the path at a specified offset. This feature will be used specify a formation with multiple agents.

2-2-3 Adaptation for unicycle robots

The previously given conversion to spatial vehicle dynamics assumes the use of a bicycle model [1]. In contrast, this thesis work uses unicycle type robots. The difference between these two modeling approaches is the presence of slip. Since a unicycle is not subjected to slip, the error dynamics can be simplified assuming $\dot{y} = 0$.

$$\dot{y}' = \frac{\ddot{y}}{\dot{s}}, \quad \dot{x}' = \frac{\ddot{x}}{\dot{s}}, \qquad \dot{\psi}' = \frac{\ddot{\psi}}{\dot{s}} \tag{2-17}$$

$$e_\psi' = (\psi - \psi_s)' = \dot{\psi} - \psi_s' \tag{2-18}$$

$$e_y' = \frac{\dot{e}_y}{\dot{s}} = \frac{\dot{x} \cdot \cos(e_\psi) - \dot{y} \cdot \cos(e_\psi)}{\dot{s}} \tag{2-19}$$

$$e_y' = \frac{\dot{e}_y}{\dot{s}} = \frac{\dot{x} \cdot \cos(e_\psi)}{\dot{s}} \tag{2-20}$$

The spatial vehicle dynamics can then discretized for use in the prediction model.

$$\dot{s}_s = \frac{\rho_s}{\rho_s - e_y} \cdot \dot{x}_s \cdot \cos(e_\psi) \tag{2-21}$$

$$e_{\psi_{s+\Delta s}} = e_{\psi_s} + \Delta s(\dot{\psi}_s - \psi_s') \tag{2-22}$$

$$e_{y_{s+\Delta s}} = e_{y_s} + \Delta s \frac{\dot{x}_s \cdot \cos(e_{y_s})}{\dot{s}_s} \tag{2-23}$$

2-2-4 Path generation

The spatial predictive controller is capable of handling any smooth path. In this thesis work a simple path generating method called Dubin's path [21] will be used. Dubin's method produces a path that consists of three either curved C or straight S parts. Paths can be either $C-S-C$ or $C-C-C$ as seen in figure 2-7. The length of the curved parts is determined by the minimal turning radius of the chosen mobile agent.

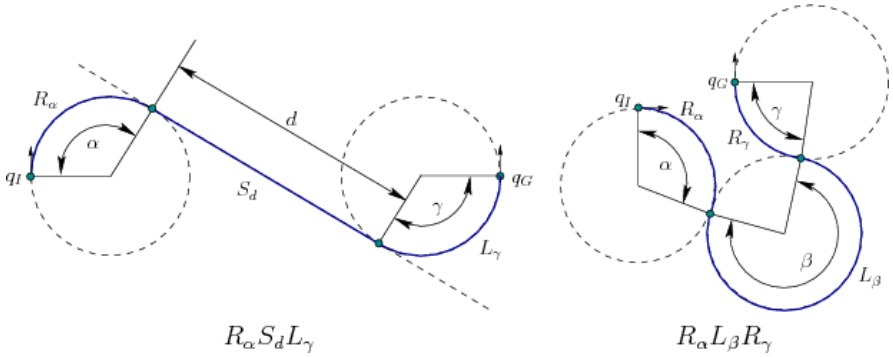


Figure 2-7: Generating Dubin's Path [2]

This path generation method relies on a vehicle model called Dubin's car with the following dynamics:

$$\dot{x} = v\cos(\theta), \dot{x} = v\sin(\theta), \dot{\theta} = w \tag{2-24}$$

Where $v \geq 0$. As can be seen, this model is similar to the unicycle but adds a constraint backward motion. The definition of a path as a set of straight and curved parts is convenient for the spatial predictive control algorithms since it can be simplified as the tracking error dynamics are simulated using the radius of curvature of the path which means $\rho = 0$ for straight parts and $\rho = R$ for curved parts where R is the turning radius of the vehicle. This allows the path to be partitioned in three segments (2-25).

$$\begin{aligned} 0 < s \leq s_1 \qquad & \rho = \rho_1 \\ s_2 < s \leq s_2 \qquad & \rho = \rho_2 \\ s_3 < s \leq s_3 \qquad & \rho = \rho_3 \end{aligned} \tag{2-25}$$

2-3 Spatial Predictive Formation Control

The previously derived Spatial Predictive controller provides path tracking for a single mobile agent with a fixed reference input. Extending this controller for formation control requires choosing these reference inputs such that mobile agents keep a predefined distance to each other whilst tracking the path. These reference inputs will be determined using a consensus approach based on which agents in a formation have the ability to share information. Then the final control structure will be defined.

2-3-1 Formation definition

Formation control methods, as specified before, use distance and heading angle constraints between agents to construct a formation.

$$\begin{aligned} d_{ij}^2 &= (x_i - x_j)^2 + (y_i - y_j)^2 \\ \Delta\theta_{ij} &= \theta_i - \theta_j \end{aligned} \tag{2-26}$$

Using these constraints yields a rigid formation, implying that the formation maintains its shape while tracking the reference path. In this thesis work, in contrast to the rigid formation definition, the distance keeping constraint between agents will be defined as (2-27).

$$\Delta s_{ij} = s_i - sj \tag{2-27}$$

This implies that agents will keep a specified distance between each other with respect to the reference path, similar to cars driving behind each other on a road. Furthermore, agents will be given an offset to the path center by defining $e_{y_i} = d_i$.

The SPC method requires that an agent has access to the reference path. Reviewing Section 2-1, a formation of agents with single integrator dynamics with access to a time-varying reference state reach consensus using (2-28).

$$\dot{x}_i = u_i = g_{i(n+1)} f(t, \xi^r) - \sum_{j=1}^{n} g_{ij} k_{ij} (\xi_i - \xi_j) - g_{i(n+1)} \alpha_i (\xi_i - \xi^r) \tag{2-28}$$

As can be seen, the velocity for each agent is determined based on the path tracking error and distance error between agents that have an information sharing connection. This exact approach will be used to extend the Spatial Predictive Control method by calculating reference velocities $\dot{s}_{\text{ref;i}}$ along the path that balance formation keeping and path tracking.

$$\begin{aligned} \dot{s}_{ref;i} &= u_i \\ u_i &= \dot{s}_{\text{ref;form}} - \Sigma_{j=1}^{n} g_{ij} k_{ij} (s_{ref;i} - s_{ref;j}) \end{aligned} \tag{2-29}$$

Where g_{ij} is a scalar weight, and $k_{ij} = 1$ if an agent i and j are connected, and otherwise $k_{ij} = 0$.

2-3-2 Formulation of Optimal Controller

The optimal control problem for path following given the spatial error dynamics for the unicycle is defined in (2-30). The cost function aims at minimizing the path deviation e_y, angle deviation e_ψ, reference path velocity $\dot{s}_{ref}$ and inputs U and ΔU. Each agent within the formation chooses its path reference velocity based on the consensus algorithm.

$$\min_u \sum_{k=s}^{s+Hp} \|e_{ref;y;i} - e_{y;i,s}\|_{Q_1}^2 + \|e_{\psi;i,s}\|_{Q_2}^2 + \|\dot{s}_{ref;i} - \dot{s}_{i,s}\|_{Q_3}^2 + \|U_{k,s}\|_{R_1}^2 + \|\Delta U_{k,s}\|_{R_2}^2 \tag{2-30}$$

The cost function has a prediction horizon of $H_p \Delta s$ meters where Δs is the discretization step size.

The final controller combining the formation control and tracking control step consists of multiple parts. First, for a known starting and ending pose x_0 and x_f a path is generated using Dubin's method. This path is computed centrally to ensure each agent within the formation tracks the same path. Then, using the path and reference velocity each agent computes its optimal control problem and executes the calculated input. The updated state is shared with other agents within the formation according to the predefined communication topology.

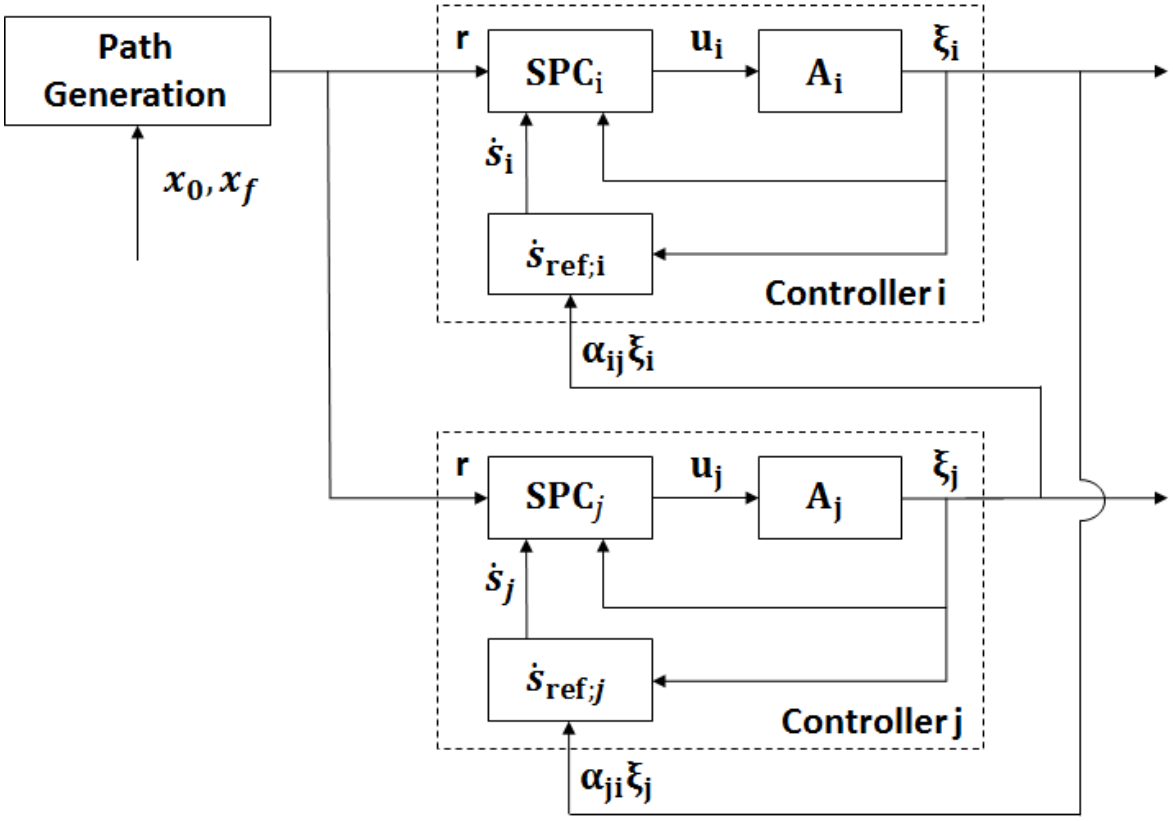


Figure 2-8: Control structure for two agents i and j

Figure 2-8 depicts the control structure for two agents i and j. As can be seen, the controllers for each agent are separate subsystems that both have access to the same path. If agent j is connect to i, $a_{ij} = 1$ resulting in information flow from agent j to i and an updated formation keeping velocity. If there is no information flow, $a_{ij} = 0$. Similarly, if agent i is connected to j, $a_{ji} = 1$ and $a_{ji} = 0$ otherwise.

2-4 Simulation results

Simulations have been performed using the Robot Operating System ROS [22] where every part of the defined controller is run as a separate software node. Details on the use of ROS are given in Chapter 3. Two different formations have been simulated consisting of 6 and 12 unicycles respectively. The path generation node, formation controller and unicycle nodes

have all been implemented in Python. For n agents, $2n + 1$ software nodes are launched. Each node is executed as an independent program. The unicycle simulation nodes use a sampling time of 20 Hz and publish updated poses at the same interval. The control nodes calculate a new control input at a rate of 10 Hz. Since every node is run separately, these processes are not synchronized and therefore are not guaranteed to publish control inputs or poses simultaneously. Python code for the unicycle simulation is found in the Appendix in Section B-1 and for the controller in Section B-2, B-3, B-4, and B-5. The optimal control problem is solved using a Sequential Least Squares Programming (SLSQP) [23] algorithm as implemented in the Python library SciPy [24]. The simulations are run on a PC with specifications according Table 2-1 and the configuration of the controller parameters is given in Table 2-2.

Property	Value
Processor	4x Intel(R) Xeon(R) CPU E5603 @ 1.60GHz
Memory	4037MB
Graphics card	GeForce GTX 580

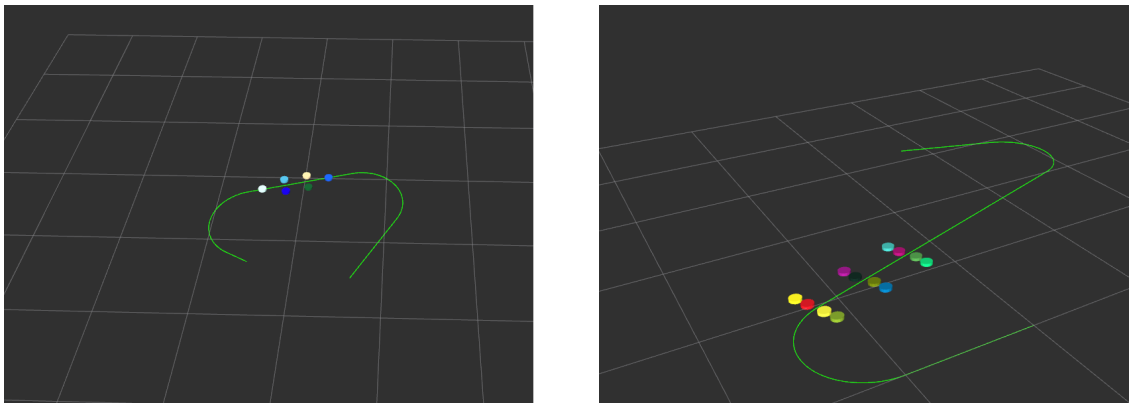
Table 2-1: Hardware specifications for simulation PC

Parameter	Value
Unicycle model sampling interval Δt	0.05 s
Controller sampling interval	0.1 s
Controller spatial step size ds	0.1 m
Controller prediction horizon H_p	0.5 m
Controller control horizon H_c	0.5 m
Controller reference velocity $\dot{s}_{\text{ref}}$	0.2 ms
Controller consensus constant g_{ij}	0.3
Control weight path deviation W_{e_y}	100
Control weight heading error W_{e_ψ}	5
Control weight reference velocity $W_{\dot{s}}$	2
Control weight input change W_{du}	1
Input linear velocity bounds v_{min}	0.01 m/s
Input linear velocity bounds v_{max}	1 m/s
Input angular velocity bounds w_{min}	-0.5 rad/s
Input angular velocity bounds w_{max}	0.5 rad/s

Table 2-2: Controller and model parameters used in simulation

2-4-1 Formation of 6 agents

The first formation experiment contains 6 nodes connected as shown in Figure 2-10. As can be seen, each node has access to the reference path as is required by the Spatial Predictive Controller. The relative positions of agents in the formation are defined with respect to the front agent as given in Table 2-3.



(a) Formation containing 6 nodes **(b)** Formation containing 12 nodes

Figure 2-9: Graphic depiction of formation control simulations

Parameter	Agent 1	Agent 2	Agent 3	Agent 4	Agent 5	Agent 6
Relative path distance Δs	0 m	-0.25 m	-0.50 m	-1 m	-0.75 m	-0.5 m
Path center offset e_y	0 m	-0.1 m	-0.1 m	0 m	0.1 m	0.1 m

Table 2-3: Formation definition

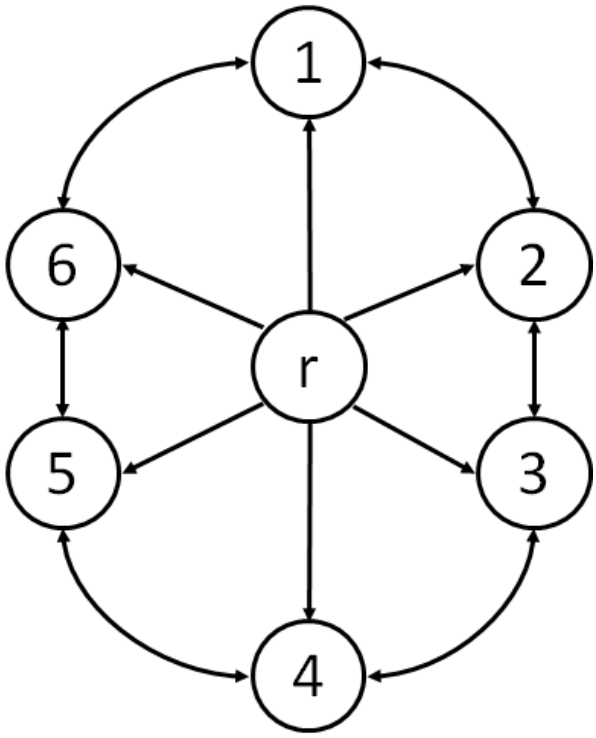


Figure 2-10: Connected, undirected graph for 6 agents

Figure 2-11 shows how the formation progresses along Dubin's path. Since the formation definition is not rigid, the formation deforms in curved path parts.

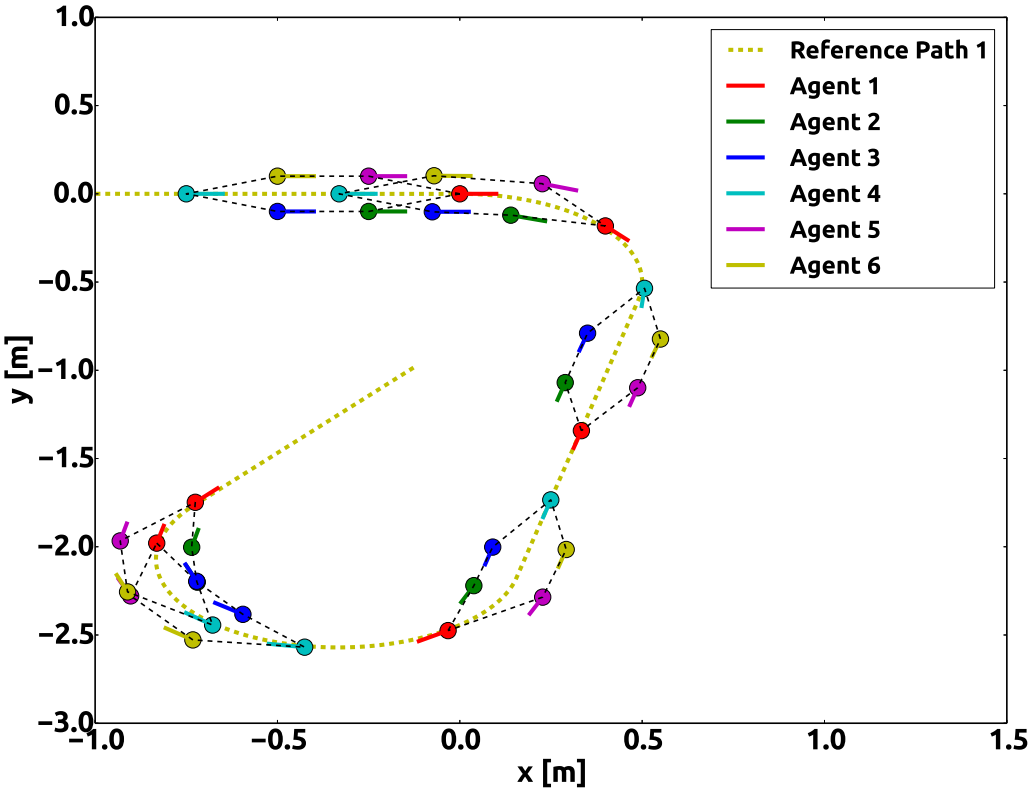
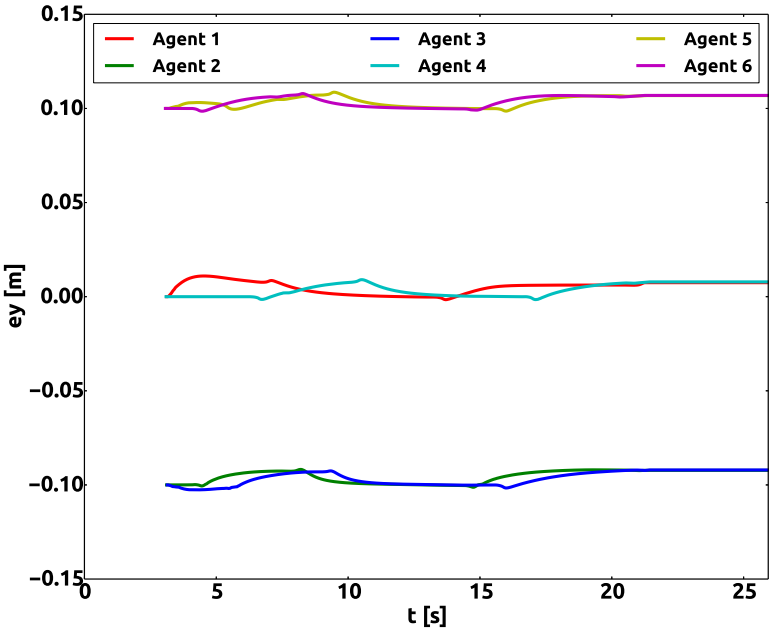
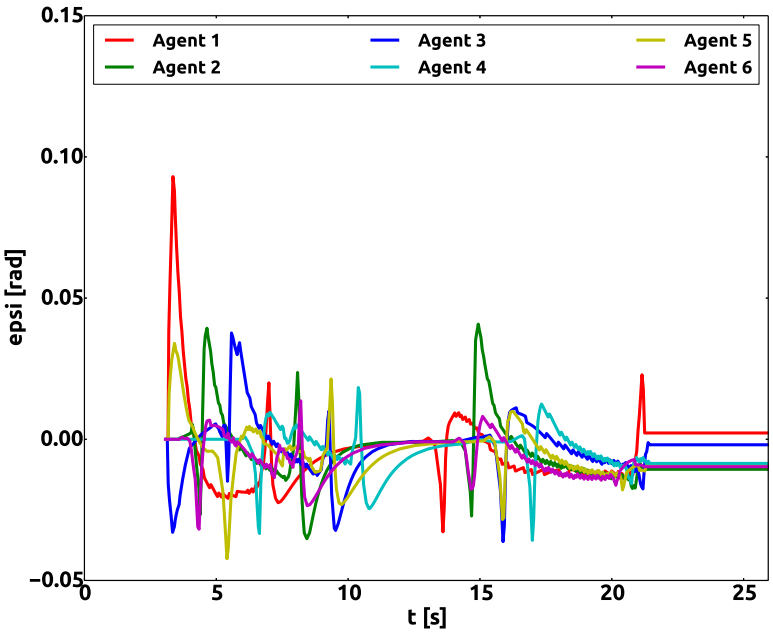


Figure 2-11: Movement of the formation in the x-y plane

Figures 2-12a and 2-12b provide the path and heading angle deviation for each agent respectively. As the formation starts moving at $t = 4$ the path error e_y can be seen to fluctuate for each agent around its reference value. For the path heading error the same can be stated, the reference being zero. Both values do not fully converge to zero. This problem is twofold. First, the optimal controller balances between tracking a given reference velocity, path heading error, path deviation and input size. The rate of convergence for each of these states can be influenced using the controller weights. For instance, increasing the weight on path deviation error might improve convergence but it would at the same time increase the heading angle error as the agent will have to change its heading angle more to reach the right offset. The controller weights for each state can further tuned to decrease the error seen in this simulation. Second, both the controller and model are implemented as separate software nodes that synchronized and run at different sampling times. Therefore, the model will not perfectly perform as simulated in the optimal controller. This mismatch is also reflected in these values. The maximum measured position error in this experiment with the given configuration is 1.099 cm and the maximum angle error 5.3°. Figures 2-13a and 2-13b show the linear and angular velocities as calculated by the controller and executed by the model. When Dubin's



(a) Path center deviation e_y



(b) Heading angle deviation e_ψ

Figure 2-12: Path tracking offsets

path is first sent to each agent, movement per agent is not initialized simultaneously. This introduces jitter at the start of the experiment as the reference velocity update algorithm produces fluctuating values for each agent. At ($t = 5$ when the formation enters the first curved path section this disturbance has been resolved and the formation is formed. Also it can be seen that the computed linear velocity is higher for agents that drive on the outer part of a curve and lower for agents that drive on the inner part of a curve. Figure 2-14 gives an overview of the formation keeping errors between the front agent 1 and agents 5-6. The maximum formation keeping error was measured to be 8 cm. As can be seen the formation keeping error fluctuates at the same instances the heading angle and path deviation values fluctuate which is when switching from a straight to curved path section and vice versa.

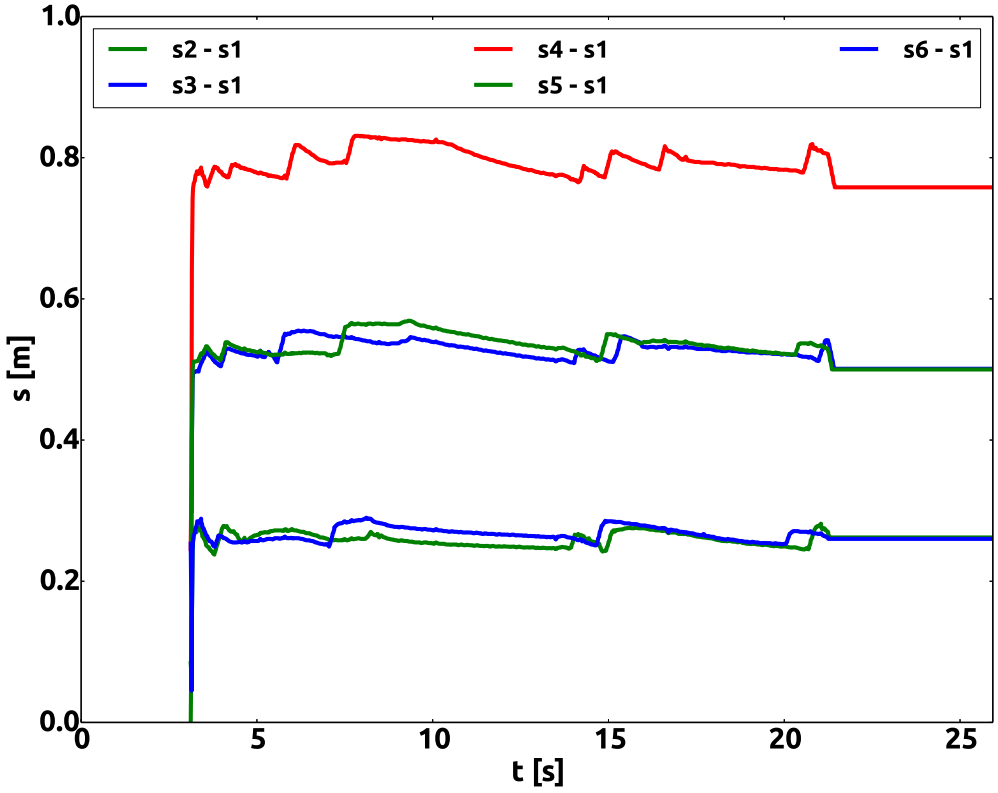
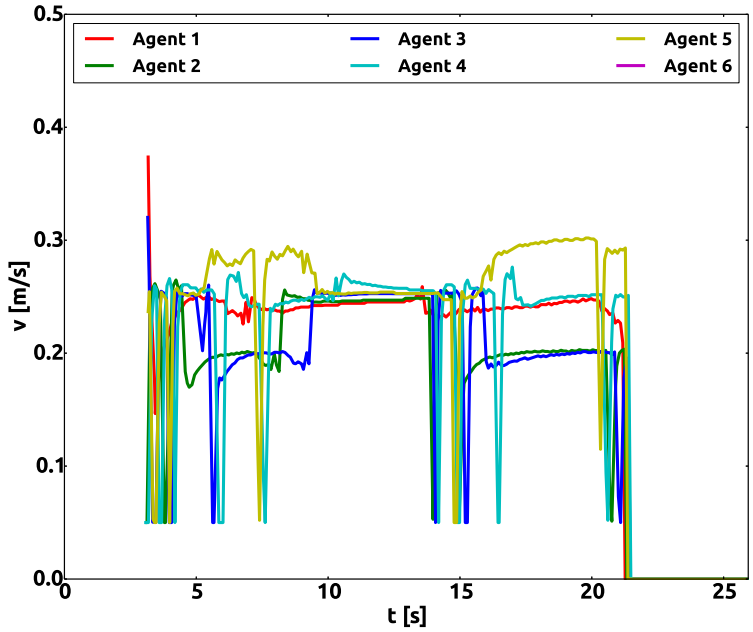


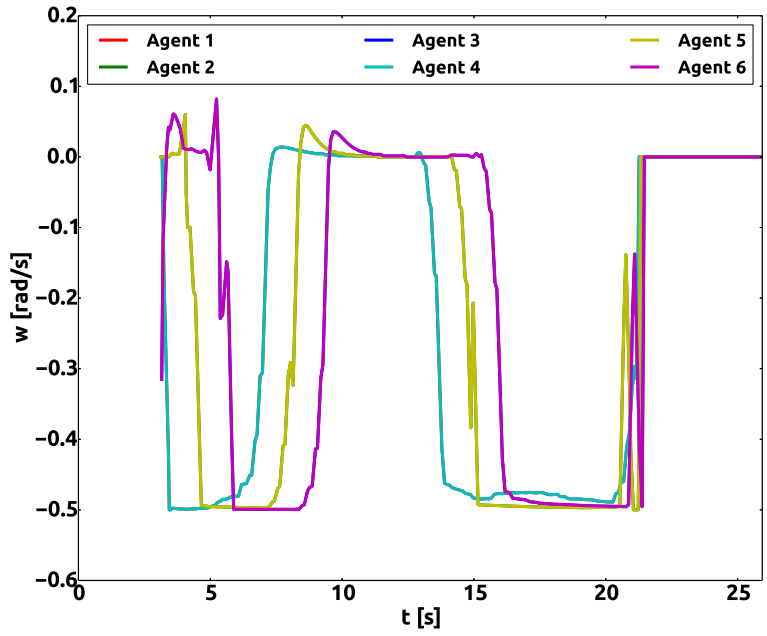
Figure 2-14: Formation keeping errors with respect to front agent

2-4-2 Formation of 12 agents

The simulation experiment using 6 agent is repeated with 12 to demonstrate the scalability of the developed control algorithm. Figure 2-15 depicts the connectivity graph and Table 2-4 provides the formation configuration.



(a) Linear velocities v



(b) Angular velocities w

Figure 2-13: Agent velocities

Parameter	Agent 1	Agent 2	Agent 3	Agent 4	Agent 5	Agent 6
Relative path distance Δs	0 m	0 m	-0.5 m	-0.5 m	-1 m	-1 m
Path center offset e_y	-0.08 m	0.08 m	0.08 m	-0.08 m	-0.08 m	0.08 m
Parameter	Agent 7	Agent 8	Agent 9	Agent 10	Agent 11	Agent 12
Relative path distance Δs	0 m	0 m	-0.5 m	-0.5 m	-1 m	-1 m
Path center offset e_y	-0.2 m	0.2 m	-0.2 m	0.2 m	-0.2 m	0.2 m

Table 2-4: Formation definition

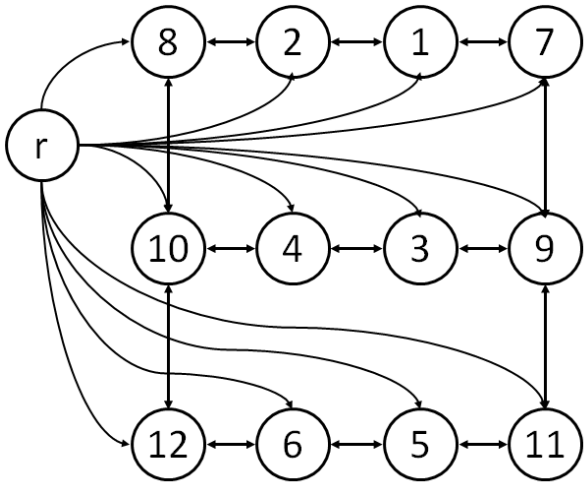


Figure 2-15: Connected, undirected graph for 12 agents

Studying Figure 2-16 it can be seen that the formation tracks the path accurately. The maximum path deviation error is found during the initialization of the experiment at $t = 5$ for agent 1 with a value of 14 cm. This offset is resolved at $t = 12$ as the formation progresses normally. This initialization error occurs due to the number of processes that are run simultaneously for this experiment. The peak in the path deviation error during the start of the experiment is also due to initialization as it is the point in time where the controller is provided Dubin's path.

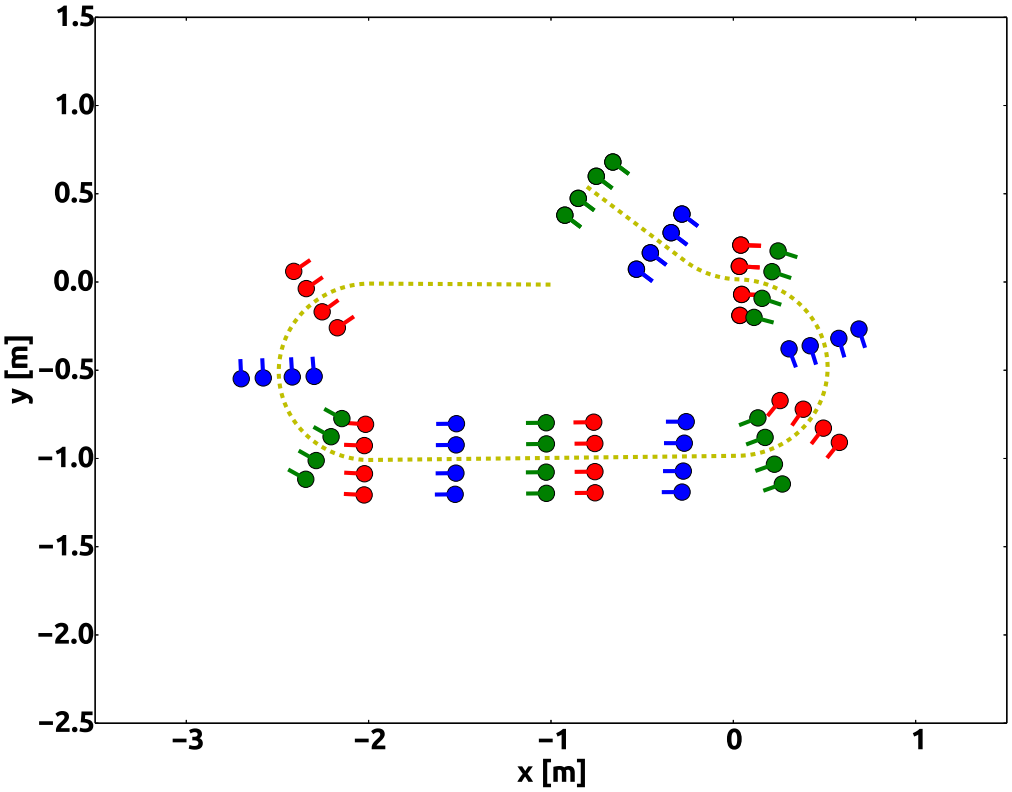
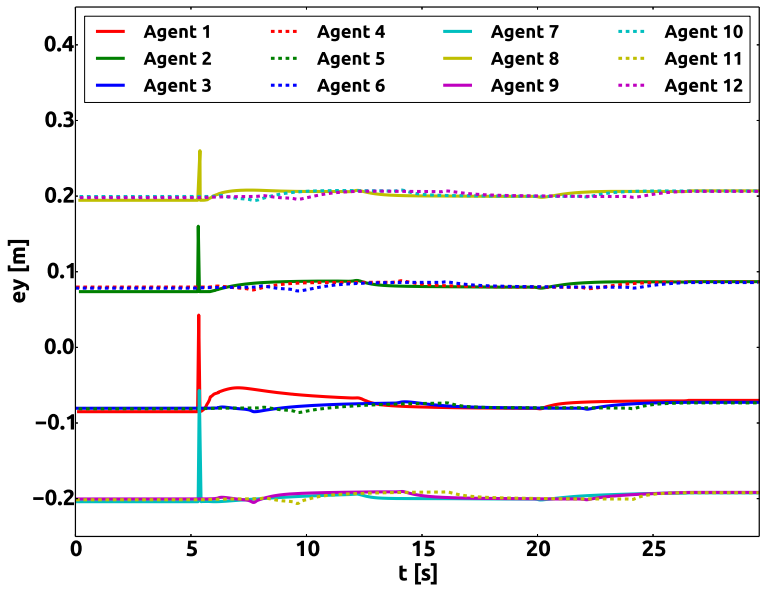
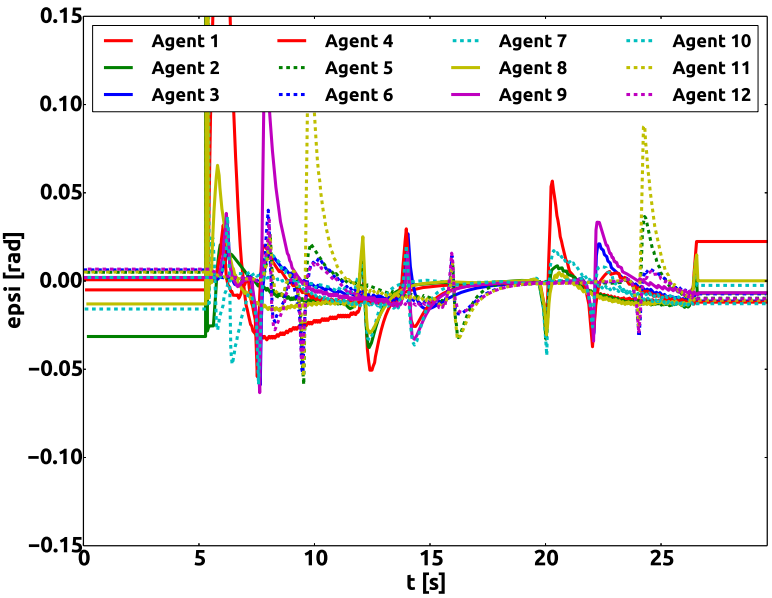


Figure 2-16: Movement of the formation in the x-y plane

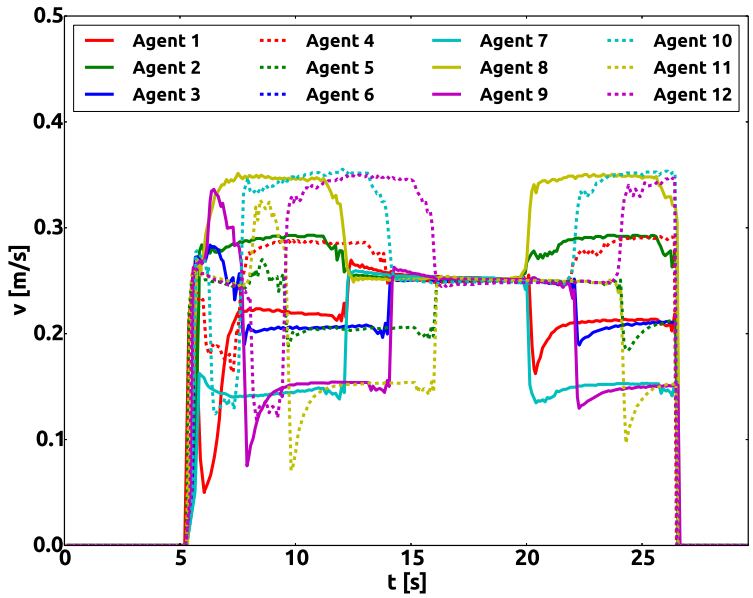


(a) Path center deviation e_y

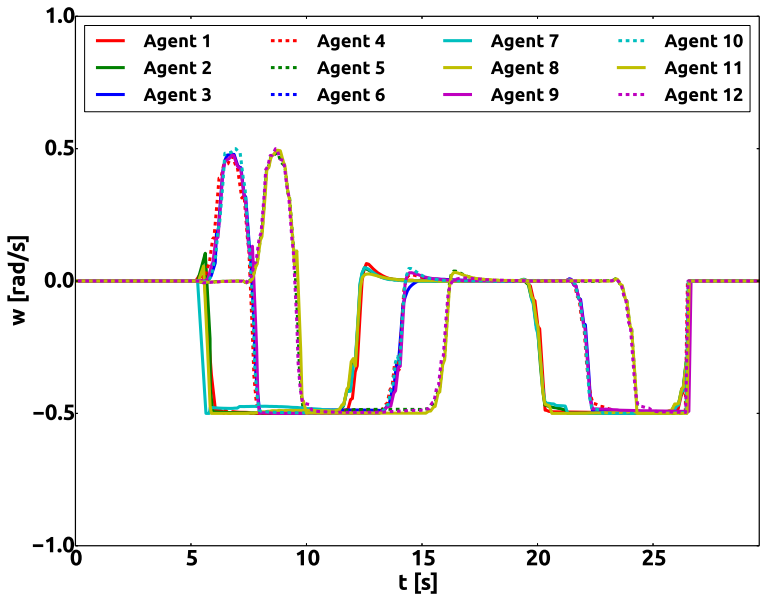


(b) Heading angle deviation e_ψ

Figure 2-17: Path tracking offsets



(a) Linear velocities v



(b) Angular velocities w

Figure 2-18: Agent velocities

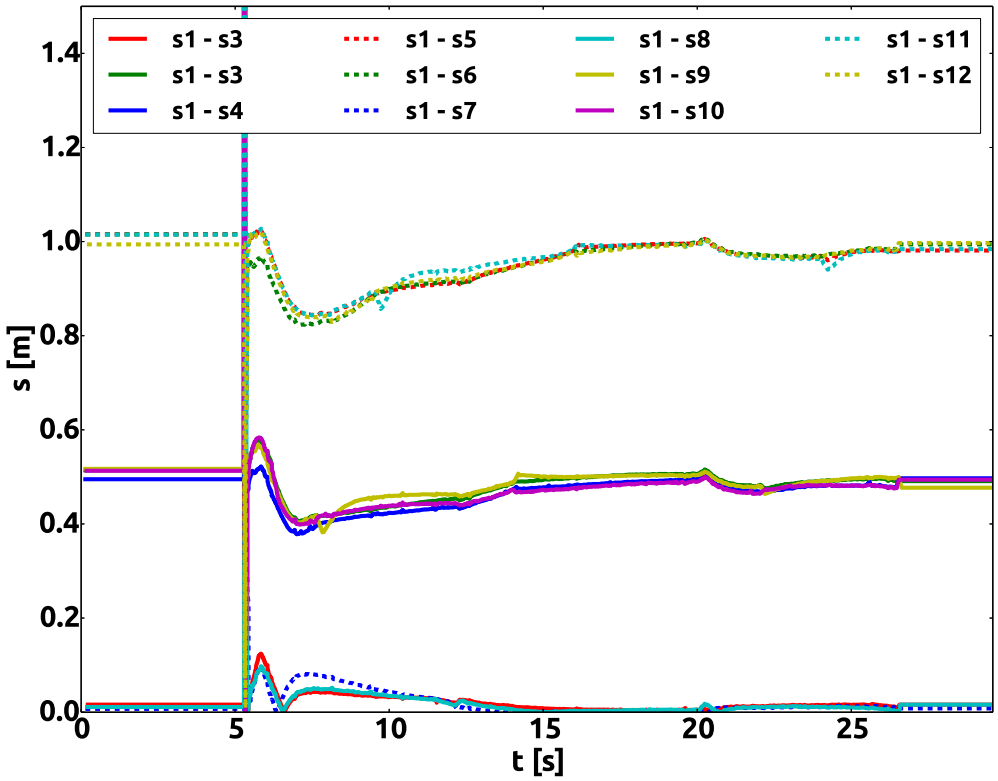


Figure 2-19: Formation keeping errors with respect to front agent

The maximum formation keeping error that occurs in Figure 2-19 is 19.9 cm which is more than double the maximum error during the experiment with 6 agents. The main reason for bigger offset can be found in the angular velocity, Figure 2-18b, which is satured at $w = -0.5$ during curved sections of the path for agents at the outside of the curve. Comparing this formation to the previous it can be seen that the path center deviation for the outer agents is double. This larger turning radius requires angular velocity higher than allowed, leading to saturation. As long as this saturation occurs, the formation keeping error grows. This shows that when selecting a formation, the path offset should be chosen such that at a desired reference velocity the input should not saturate.

2-5 Summary

In this chapter a distributed formation controller was developed by combining a consensus approach with the Spatial Predictive Control method. The resulting controller is able to achieve and maintain a formation of unicycle type robots whilst tracking a reference path within certain error bounds. The resulting controller implementation is scalable to any number of nodes as the controller does not have to run centrally and does not need access to position information of all agents.

Chapter 3

Development of the Networked Embedded Robotics Lab

The goal of the Networked Embedded Robotics Lab is to provide an easy-to-use and reliable setup promoting code reusability to make sure a student's work can be adopted by others. This chapter focuses on the software and hardware infrastructure necessary to achieve this. First an overview of robotic middleware, software libraries that contain often used functionality, will be given. Then details are provided on the implemented hardware and software infrastructure, the available robotics platforms. Last, examples are given for experiment interaction based on the lab infrastructure.

3-1 Robotic Middleware

As mentioned in the research objectives, an important feature of the lab's software infrastructure should be usability and re-usability. This involves using standardized libraries and modular frameworks. These libraries and frameworks come in the form of robotic middleware. Some of this middleware focuses on providing optimized and generalized methods for tasks like state estimation whilst others focus more modularity through standardized messaging protocols. Functionality of robotic middleware can be listed as follows [25]:

- Simplification of the development process.
- Support communications and interoperability.
- Provide efficient utilization of available resources.
- Provide heterogeneity abstractions.
- Support integration with other systems.
- Offer often-needed robot services.

- Provide automatic resource discovery and configuration.

As many robotic middleware frameworks are available, this section will be limited to some of the most used frameworks. A more detailed overview is given in appendix A.

Robot Operating System (ROS) is as at the time of writing the most widely accepted robotic middleware in the academic world. It is an open source project for distributed robotics maintained and first released in 2007 by Willow Garage [22]. Its goal is to simplify the complex process of developing robotic software. ROS does this by providing users the tools to separate large software projects into multiple smaller programs, called nodes, that can be developed and run separately. For separation of software into smaller nodes, ROS uses the publish/subscribe model. With the publish/subscribe interaction scheme, subscribers register their interest in an event and are subsequently asynchronously notified of events generated by publishers. Figure 3-1 provides an example of a typical distribution of nodes for the control of a robotic platform. The platformś driver exchanges information with the physical hardware. This implies setting control commands and reading sensor information. A simple ROS driver for a platform would publish sensor information to other nodes and would subscribe to generate control commands and send these to the physical hardware. A state estimation node would process sensor information and a controller node would use these to generate control commands. As can been seen, nodes communicate directly with each other though the register to a central node called the *roscore*. This is a node that keeps track of all other nodes running within the ROS ecosystem and allows them to find each other.

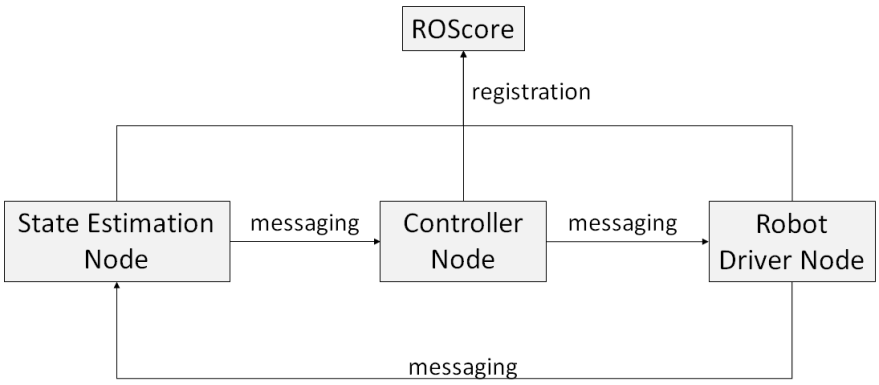


Figure 3-1: Simplified depiction of software separation in nodes

Figure 3-1 displays a simplified situation and in general, more software nodes are used. The main benefit of this publish-subscribe model is re-usability. By standardizing the information type that is shared between nodes such as state estimators, it becomes trivial to re-purpose them for new projects without any programming. By letting students in the Networked Embedded Robotics Lab uses this programming paradigm, all developed software will be highly modular. This modularity allows students to develop software to can be re-purposed or improved upon with less effort that it would normally as they can focus solely on the nodes that are important to their research. On top of the publish-subscribe model, ROS also supplies multiple tools for use in experiments such as a 3D visualization tool called RViz.Due to ROS's popularity software nodes for many types of applications and robotic

platforms are available and shared among researchers. Examples of implementations such as collision avoidance methods in [26] using quad-rotors or indoor navigation in [27] or outdoor autonomous quad-copter swarms using GPS in [28] are widely available.

Orocos, Open Robot Control Software, is a collection of C++ libraries for kinematics, dynamics and filtering. It can be used as a standalone package but is also used as a basis for other robotic middleware[29]. It was first introduced in 2001 by the University of Leuven [30]. It's released under the LGPL, lesser GPL, license. Examples of applications are autonomous cars, compliant motion task specification, visual servoing and 3D motion tracking. Orocos consists of three main features: a Bayesian filtering library, a kinematics and dynamics library and the Orocos toolchain. Orocos uses C for real-time development, C++ for higher lever algorithms, XML for configuration purposes, Modelica for modeling and CORBA [31] for communication between software components. The Bayesian filtering library [32] provides methods such as (extended) Kalman filters and Particle filters. The kinematics and dynamics library (KDL) provides methods for modeling and controlling kinematic chains found in complex multilink robotics. Motion can be defined by convenient geometrical objects and coordinates can be easily be converted between coordinate reference frames. In [33] the KDL library is used to derive Jacobians for two robotics arms and let them perform a cooperative task with several constraints as well as keeping track of people in the vicinity. The Orocos toolchain is a tool to design real-time applications using a modular approach. It also provides extensions to other robotics frameworks such as ROS, Rock and Yarp which will be covered later. The most interesting component is the Real-Time Toolkit (RTT). The RTT provides OS abstraction, scripting tools and communication tools with a focus on real-time behavior allowing for lock-free data exchange. In [34] Orocos is used to design a real-time controller for a robotic manipulator.

The Player project is a robot simulation suite, allowing to move simulated as well as real robots through a virtual environment. It is one of the first major robotic middleware frameworks and on of the most used. There is support for a large number of robots and it is a popular software suite. Development started at the University of Southern California and was released on Source Forge in 2001 under the GPL license [35, 36]. Player is, in essence, a very minimal robotic framework. It provides a communication method between several components of a complex software system. Components such as actuators, sensors and robots are all clients in Player where Player itself acts as a server to all communication needs. Clients can be written in virtually any software environment that allows the use of TCP sockets and can be as complex or easy as desired. A distinction can be made between drivers, communicating with hardware directly providing specific code for a specific robot base or sensor and interfaces which process driver data. The device driver might, for instance, provide simplified information such as a robot's position or a distance measurement from a sensor while a client can use this data to perform operations. Drivers could also be virtual for simulation purposes [36]. By making this separation interface code can remain unchanged when swapping hardware. A large set of robotic platforms and sensors is supported within the Player project. With the release of Player 2.0 several improvements have been made [37]. Communication can now be performed using CORBA, JINI and shared memory on top of using TCP and uses a queue-based communication system for each Player driver. Also originating from Player are Stage and Gazebo which are 2D and 3D simulation environments respectively. They can interact with Player or other frameworks (such as ROS) and be used to simulate experiments as well as visualize and control real robotic agents. It is also possible to combine virtual

agents with real agents for instance letting a user control a virtual leader robot which real robots follow.

Due to the widespread use of ROS, it has been chosen as the primary framework on which this lab is built. ROS support is available for all components in the lab. In this section, all used software and code that is used will be referenced. As new versions of ROS are released often, ROS Indigo [38] has been chosen as at the time of writing it is the best supported and recommended version.

3-2 Layout of the Networked Embedded Robotics Lab

The networked embedded robotics lab was constructed within an area of 4300mm wide, 9500mm deep and 3600mm high. Just outside this area, desks with multiple PCs and control hardware were placed.

Figure 3-2: View of the lab area with desks and PCs

Around the perimeter of the ceiling, a camera system for robot tracking needed to be mounted and a design was made as seen in Figure 3-3. A metal gutter, as often used in stores, was chosen and implemented as shown in Figure 3-4. This type of gutter is wide and allows the transport of cables as well as mounting the cameras and several other peripherals.

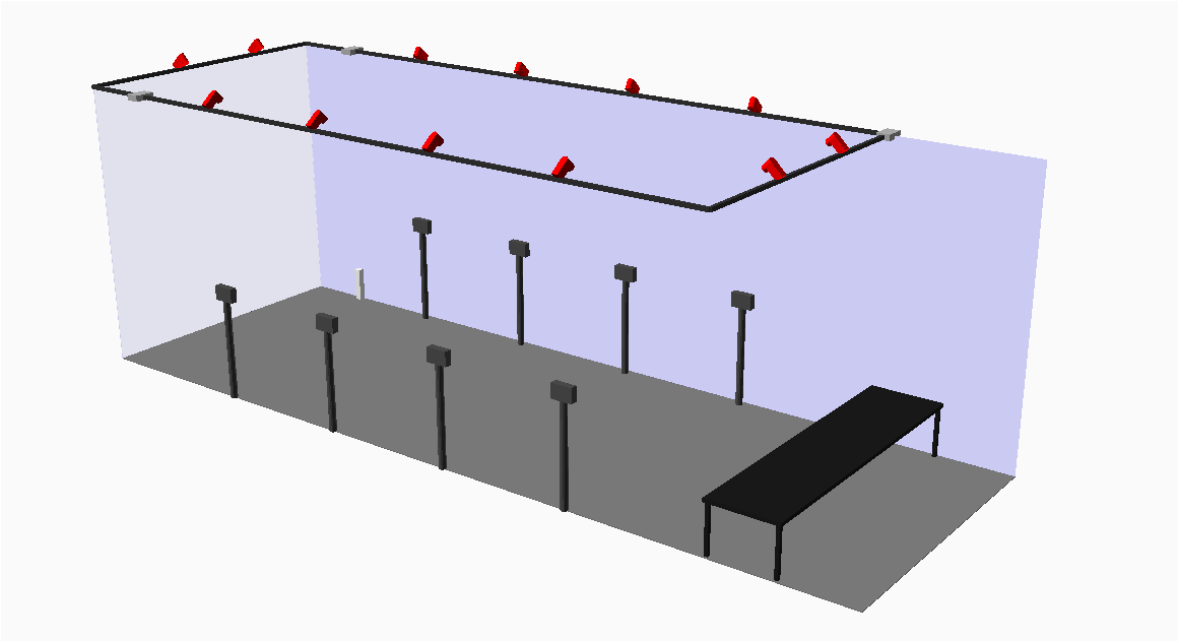


Figure 3-3: Design of camera placement in gutter

Figure 3-4: Gutter around the lab perimeter with mounted cameras

Finally, since the lab environment is to be used for aerial robots, a safety net needed to be installed. To this end, rails were installed on the bottom of the metal gutter, to supported a net that can be easily deployed and removed. Figure 3-5 shows the net within the lab environment. The net is made from two parts that can be attached to each other using velcro.

Figure 3-5: Deployable safety net for drones

3-3 Currently available robots

The Networked Embedded Robotics Lab currently provides five different robotics platforms for experimental use ranging from ground based to aerial platforms. This section provides a short description of each of these platforms and the methods used to control them.

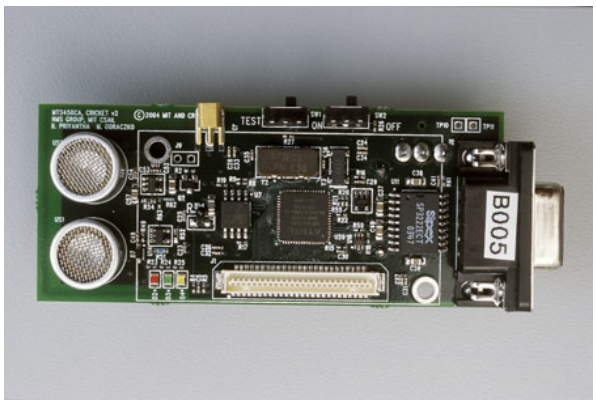
3-3-1 Create

The lab setup hosts mobile as well as aerial robotic platforms, but this thesis work focuses on the Create [39] by iRobot, a variant of the popular Roomba, platform. The Create is a programmable differential drive robot. For localization purposes, it uses rotary encoders on the wheels and has an IMU onboard. Furthermore, it has a front bumper that serves as a tactile sensor allowing for very basic mapping. Lastly, it has an IR detector on the front that can detect virtual IR walls.

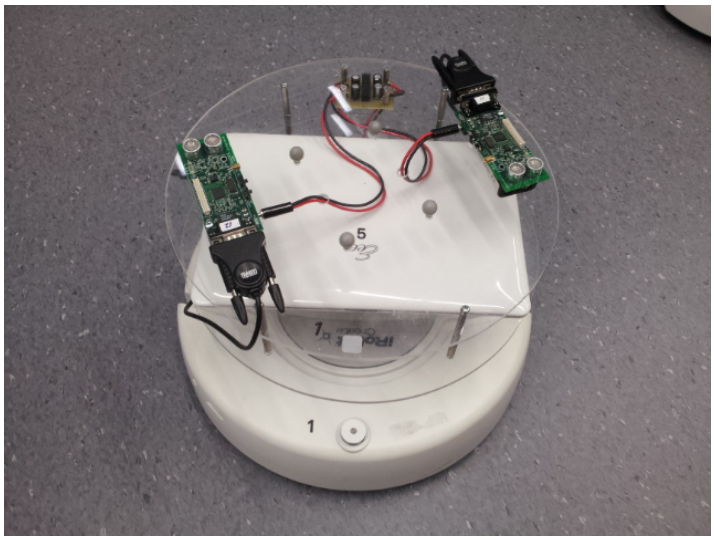
On top of the Creates, a small laptop (netbook) is mounted, which can be used to run control algorithms, and two Cricket sensors by MIT [42] for self-localization. Cricket sensors employ RF and ultrasonic communication to determine the distance between two or more units. It first sends an RF signal to other units indicating that it will send an ultrasonic signal. Units detecting this will then wait until they receive this ultrasonic signal and determine the time difference of arrival (TDOA). This translates directly a into distance. Combining the distance to multiple Crickets with known locations, set up around the perimeter of the lab area, allows a Create to determine its own location. The Create can be controlled remotely by sending commands over Wi-Fi to the attached laptop on a custom network to minimize interference.

Controlling the Create is done using the ROS control library for the popular TurtleBot [43], a commercial robotic platform built on top of a Create which is mounted with a netbook and camera. This control library accepts velocity inputs in the form of a Twist, six dimensional velocity vector, and is actively maintained and provides a stable code base to work on [44].

(a) The iRobot Create [40]



(b) Cricket sensor [41]



(c) iRobot Create with Cricket Sensors and Netbook

3-3-2 Sphero

The Sphero by Orbotix is a small, rolling robot. The Sphero features a Segway-like robot in a plastic spherical shell. These robots have various onboard sensors to measure its motion and collisions. The Sphero has been used in the lab are for Simultaneous Localization and Mapping (SLAM) experiments [45] to build a map of a rectilinear environment using only collision detection. Communication with a Sphero is done via Bluetooth. In experiments up to six Spheros were controllable simultaneously through a single Bluetooth connection.

Figure 3-7: Sphero by Orbotix [3]

The Sphero is used within the lab enviroment with unofficial ROS drivers referenced at the Orbotix development website [46]. Using this software multiple units can easily be connected and controlled using velocity input commands similar to the TurtleBot drivers.

3-3-3 Hovercraft

Hovercrafts are interesting platforms for transportation as they can cover both ground and water. In the Networked Embedded Robotics lab, research is done towards the control of a formation of low-cost hovercraft platforms in order to collaborate in the transportation of large materials. The model available in the lab is produced by Ikarus. Originally these units are controlled through RF, but these units have been outfitted with Arduinos with Wi-Fi communication. Software has been developed to control them via through TCP/IP, allowing control through ROS and Matlab. Experiments have been conducted in waypoint tracking for the hovercraft.

Figure 3-8: Hovercraft

3-3-4 Parrot AR

Parrot AR Drones are popular low-cost drones available in the lab. These drones provide an interesting platform to simulate various space type scenarios. The Parrot AR Drone creates a custom Wi-Fi network which can be connected to for remote control. Extra Wi-Fi adapters are available in the lab area to connect to these drones. Previous research with these platforms has been done in controlling the position of a slung mass [47].

Figure 3-9: Parrot AR Drone 2.0 [4]

In order to safely perform experiments with drones, deployable nets have been installed to prevent drones leaving the lab area.

Figure 3-10: Net deployed in front of PCs

3-3-5 Matrice DJI 100

The DJI Matrice is a more heavy duty drone than the Parrot. This drone provides stable flight and its functionality can be extended by adding more sensors. The platform is new to the lab and has not yet been used in practical experiments.

Figure 3-11: Matrice DJI 100 [5]

3-4 Localization hardware

An IR camera system by OptiTrack will be used [48]. A total of 16 Flex 13 camera's will be available for external localization. This will allow experimental validation of localization algorithms or for control of robots which cannot perform self-localization. The camera system

Figure 3-12: OptiHub [6]

will be connected to a PC by 3 OptiTrack hubs. This PC will run only the software needed for this camera system.

Figure 3-14 shows an overview of the interconnected components.

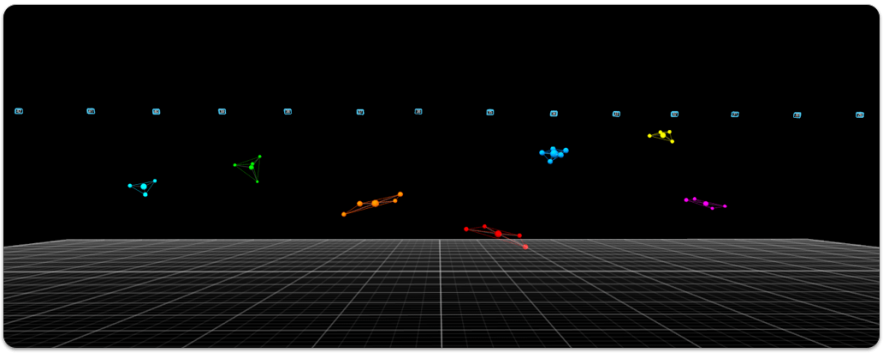


Figure 3-13: Motive tracking software [7]

3-5 Laboratory architecture

The previously mentioned platforms are all tied together through a single architecture providing communication methods, localization options and control options. In Figure 3-14 the complete hardware setup is shown. As can be seen, the Optitrack Cameras are connected to a dedicated Windows PC running the OptiTrack software, Motive, to gather and stream agent poses to all machines in the network. This is done via the OptiTrack, VRPN, or Trackd streaming protocols. This data is collected on the ROS Host PC where a ROS node, *mocap_optitrack*, converts it into a ROS topic for each separate agent. From the ROS host PC communication is established over ROS to each Netbook where a driver is running for

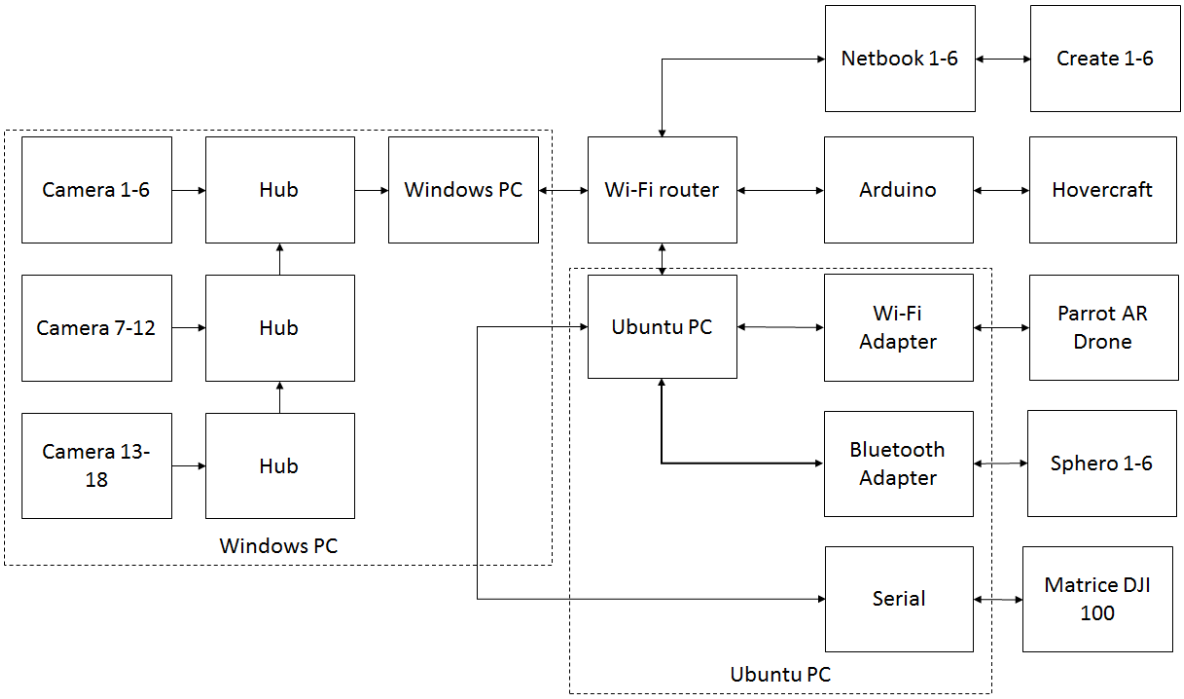


Figure 3-14: Proposed hardware setup for the lab

basic velocity tracking on the Create. Attached to the Netbook, as seen in figure 3-6c, are two Cricket sensors. These sensors communicate to Crickets installed around the perimeter of the lab for an alternate localization option to the Optitrack system. Figure 3-3 shows an example of the cameras and Crickets installed around the perimeter of the lab. Since both the netbook and host PC run ROS, it is required to synchronize their clocks. For this, Chrony [49] is recommended by ROS. The host PC is configured as an NTP server and the Netbooks synchronize their time.

3-6 Development options

ROS provides several choices in term of programming languages and development software. By default, the ROS library supports development in C++ and Python. For students that wish to use C++, Eclipse and CodeBlocks have been pre-installed. For students that prefer Python, Spyder has been installed. As of Matlab 2015b, ROS is also natively supported with the Matlab Robotics Toolbox. This toolbox provides all ROS functionality. Matlab can be used to create a ROScore and publish or subscribe to running ROS nodes. This allows students who do not wish to deviate from Matlab to seamlessly integrate their software with all previous developments.

If experiments are still in the simulation stage, students can make use two different software packages. First ROS provides the RViz package which is able to render markers and simple shapes to show the state or pose of a robot. Second, the popular Gazebo is installed, a graphical simulator complete with physics simulation. For different robotic platforms, models can be found online and run within Gazebo. Gazebo then exposes ROS topics that can be

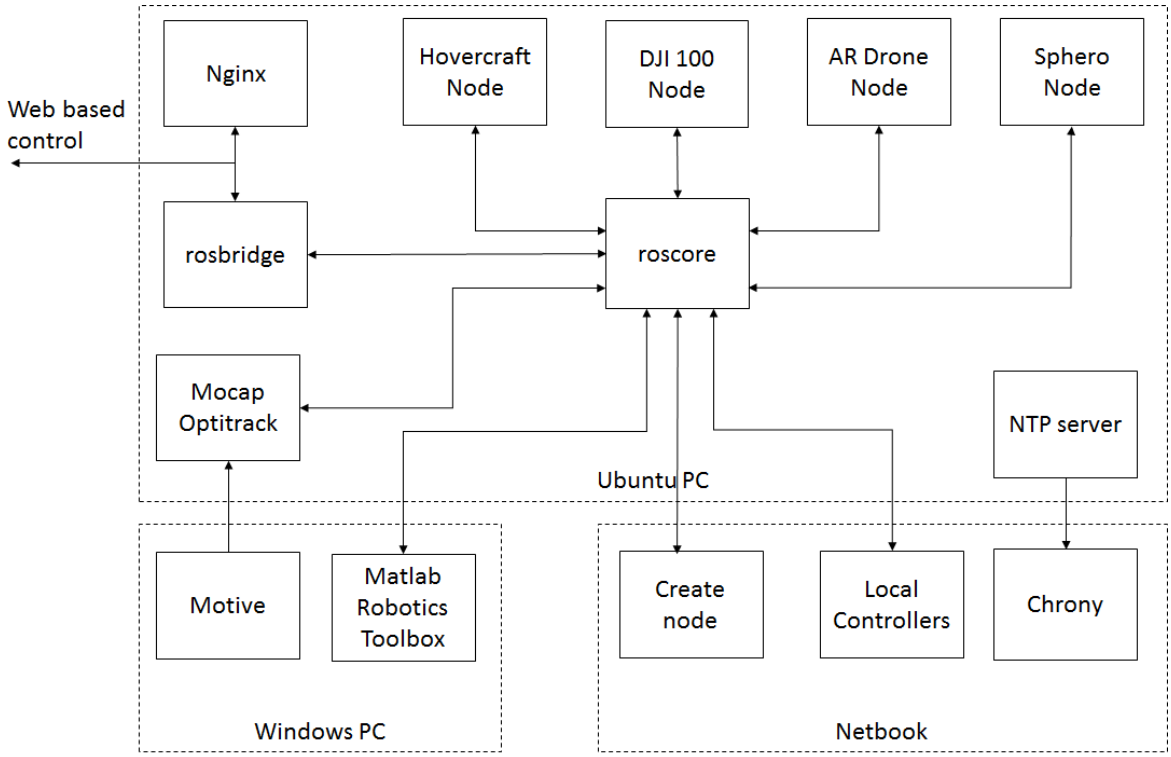


Figure 3-15: Proposed software setup for the lab

hooked into in order to create a simulation of an experiment with using the actual robotic platform or building a custom simulator. If the experiment works, it can be ported to the actual platform just by running the driver node of the robot itself rather than Gazebo.

3-7 Experiment interaction

The DCSC Networked Embedded Robotics Lab currently provides several methods to interact with experiments:

- Using a rosnode to calculate and send inputs directly to a controller.
- By using the rosbridge_server [50] package and sending input commands through websockets.
- By interfacing rosjoynode and using a game controller te remotely control experiments. A demo has been built by building a small software node that takes input from the joynode topic and sends velocity commands to the sphero cmd_vel topic.

Figure 3-16: Logitech Gamepad for remote control of agents

3-8 Web-based interactive manual

Since the *rosbridge_server* package provides web socket access to all software nodes running in ROS, it is possible to not only create interactive experiments, but also an interactive manual or guide to working with the lab setup. An initial version of this has been created for the current lab setup. A web server has been installed on the ROS host PC running Nginx as a web server configured with PHP. Using a popular PHP web framework called CodeIgniter a simple website has been developed showcasing some of the platforms available in the lab and some interactive guides have been built to start working with the lab setup. These guides include a basic tracking control experiment with a single Create, a formation control experiment with multiple Creates and a demo experiment to work with the Sphero. Each experiment goes through the commands in Linux needed to open a terminal, starting a *roscore* and launching an experiment. Students are then expected to investigate the working of these files as to be able to create their own versions of other experiments and to get acquainted with ROS.

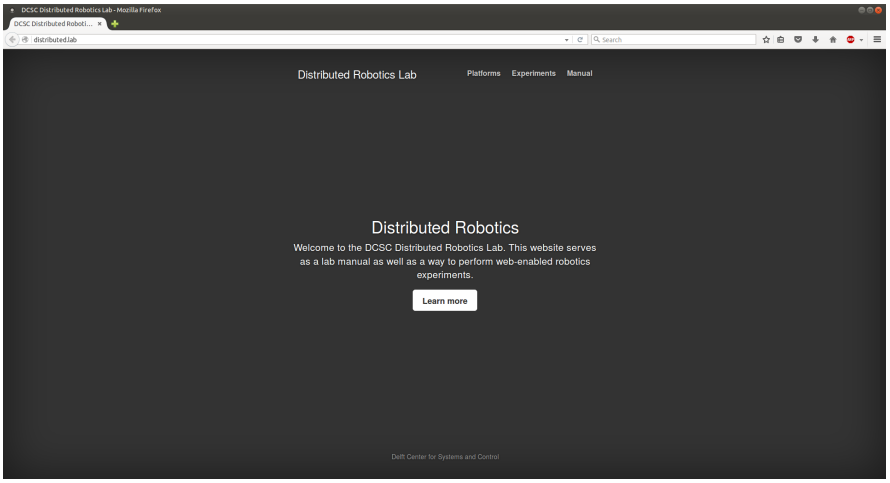


Figure 3-17: Starting page of the interactive lab website

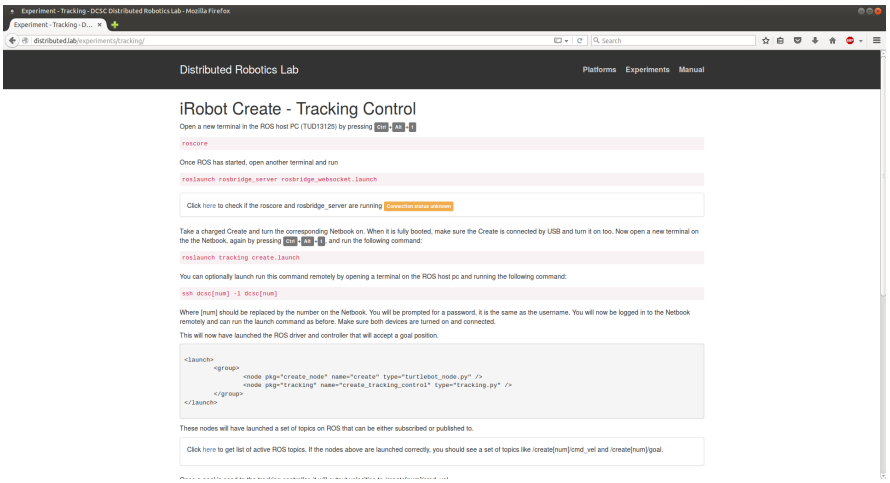


Figure 3-18: Example of an interactive experiment page for tracking control

Chapter 4

Experimental demonstration of Distributed Spatial Predictive Formation Control

4-1 Implementation aspects

The algorithm as described in Chapter 2 is implemented as a series of ROS nodes, all written in Python. For basic control of linear and angular velocity of the Create, the ROS drivers for the Turtlebot have been used. The Turtlebot is an often used robotic platform used with ROS based on an iRobot Create, a netbook and Kinect camera by Microsoft. With the exception of the Kinect, this solution is very similar to the implementation as used in the Networked Embedded Robotics Lab. On the ROS host PC, a *roscore* node is active, the *mocap_optitrack* node to distrbute agent's poses over ROS and a *rosbridge_server* node for web control. Based on the Dubin's path generator developed [51] for Python, a lightweight central path planner node has been developed that takes a goal pose for the formation of Creates from the web control panel and computes a path which is submitted to control nodes running on the netbooks for each Create respectively. These control nodes take the current pose of their Create from *mocap_optitrack* and convert it to a spatial pose by finding the closest point on the path computed by the central planner. This spatial pose is given to other controllers connected on the graph and is used for the optimization routine. A complete overview of the software nodes is generate by using the *rqt_graph* package and can be seen in Figure 4-1.

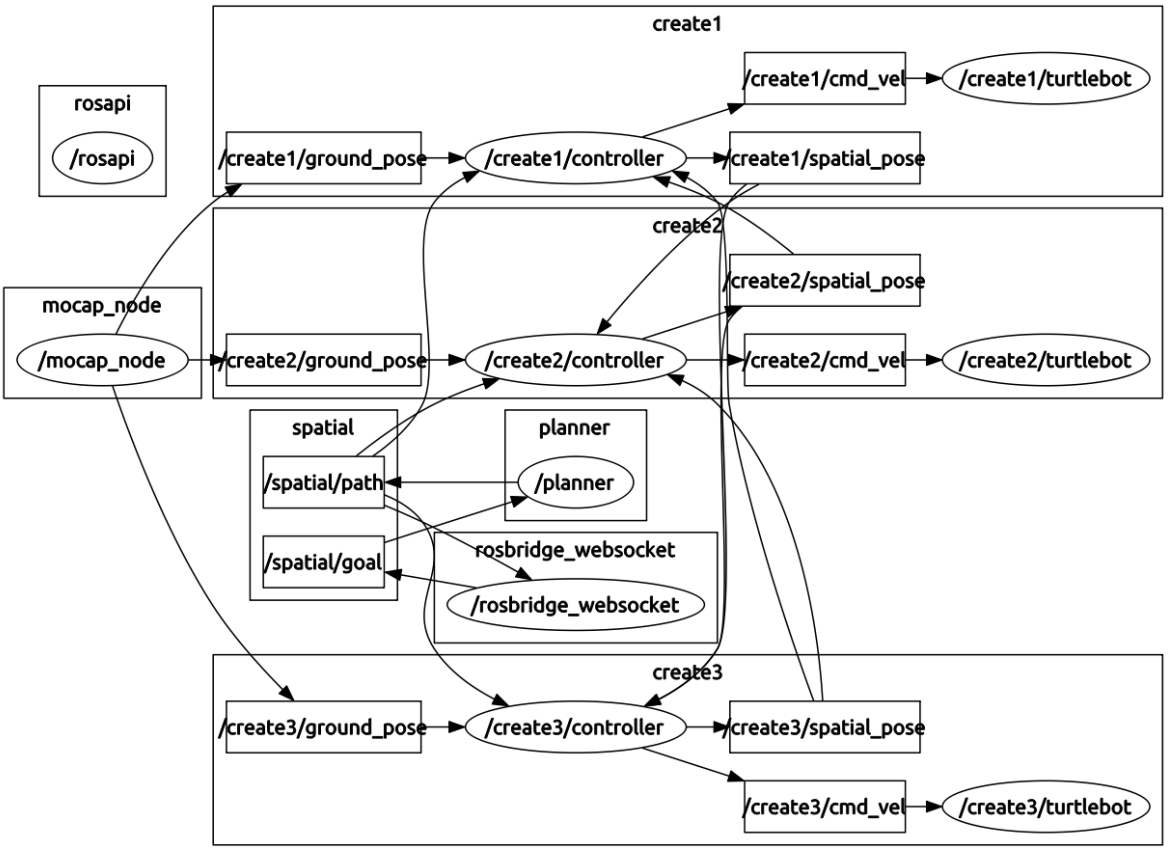


Figure 4-1: ROS node graph for distributed formation control experiment

Parameter	Value
Unicycle model sampling interval Δt	0.05 s
Controller sampling interval	0.1 s
Controller spatial step size ds	0.1 m
Controller prediction horizon H_p	0.5 m
Controller control horizon H_c	0.5 m
Controller reference velocity $\dot{s}_{\text{ref}}$	0.2 ms
Controller consensus constant g_{ij}	0.3
Control weight path deviation W_{e_y}	100
Control weight heading error W_{e_ψ}	5
Control weight reference velocity $W_{\dot{s}}$	2
Control weight input change W_{du}	1
Input linear velocity bounds v_{min}	0.01 m/s
Input linear velocity bounds v_{max}	1 m/s
Input angular velocity bounds w_{min}	-0.5 rad/s
Input angular velocity bounds w_{max}	0.5 rad/s

Table 4-1: Controller and model parameters used in simulation

4-1-1 Latency analysis

An important aspect in the practical implementation of this experiment is the measurement of communication delay or latency between different software nodes and hardware. High latency can severely limit performance if not accounted for. Reviewing Figure 4-1 there are several nodes that require information from other nodes, all these transitions introduce communication delay to the experiment either due to network latency due to communication between devices or internal latency due to the overhead of communication through ROS. Figure 4-2 shows a histogram for latency due to ROS measured on a local system. Figure 4-3 provides the same measurement but between ROS nodes on different devices within the network. These measurements provide a basis for the analysis of total latency within this experiment. The latency between nodes on the same or on different devices on the network was measured by sending a ROS message from one node to another, bouncing it back, and measuring the time the two-way communication took. The latency for one-way communication is estimated to be half that time.

The latency introduced by separating ROS nodes can be seen to be approximately 1 ms and the latency from ROS node on another device within the network was measured to be 4 ms. In contrast, the controller in this experiment employs a sampling time of 100ms. The maximum measured latency values however were 11.55 ms and 73.3 ms respectively. These values are likely caused by the fact the ROS is not a real-time framework. If real-time performance is required, it is based to allocate all critical real-time code in a single software node.

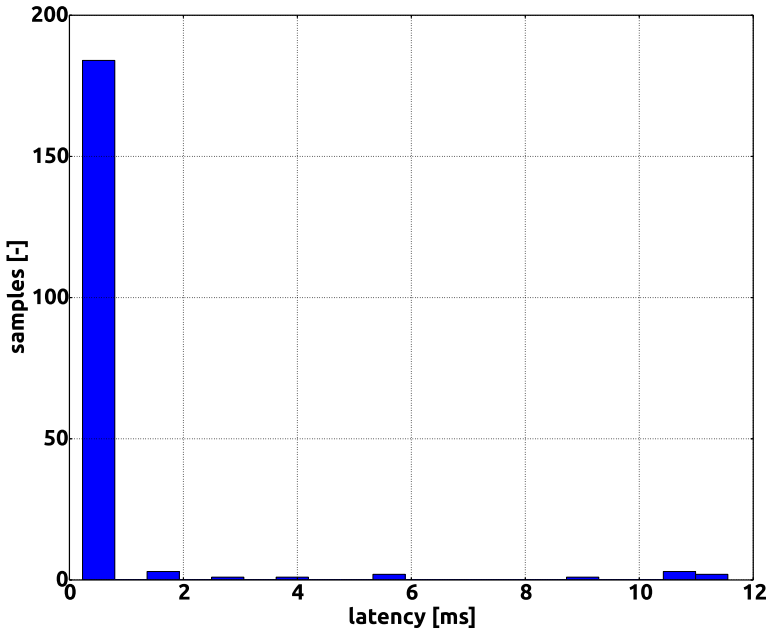


Figure 4-2: Local latency measured with 197 samples

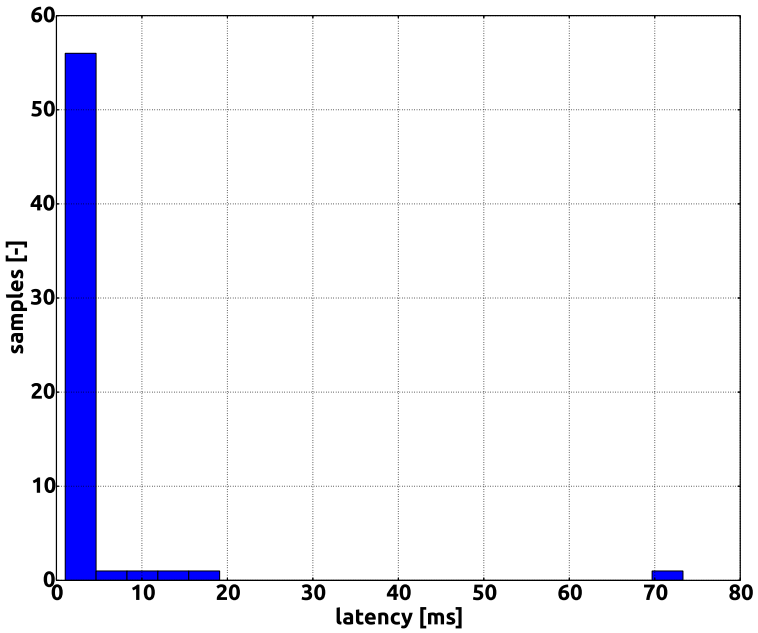


Figure 4-3: Network latency measured with 61 samples

4-2 Experimental results

In the practical experiment a triangular formation of three Creates has been tested using the Spatial Predictive controller. The formation controller was tested under normal conditions and under the condition of agent failure. The connectivity graph for these experiments is given in Figure 4-4

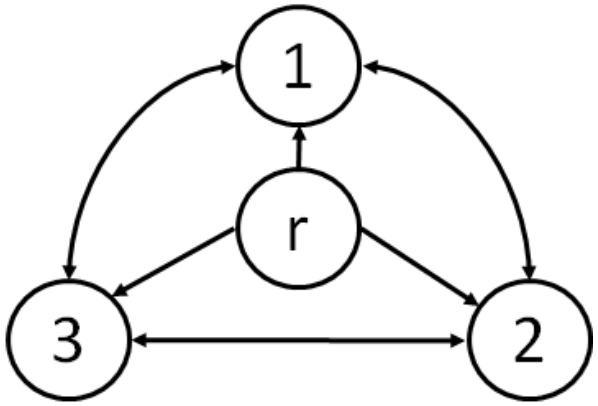


Figure 4-4: Connected, undirected graph for 3 agents

4-2-1 Formation of 3 agents

The triangular formation features three Creates. One agent moves along the path center while the two others keep a distance of $\Delta e_y = \pm 0.25$m of the center and stay $\Delta s = 0.6$m behind the front vehicle. The Creates communicate on a fully connected graph. As can be seen in Figure 4-6a the maximum path distance error is 7cm which is consistent with the simulations. the maximum distance keeping error is found to be 6.48 cm. For reference, the Create has a body diameter of 33.8 cm.

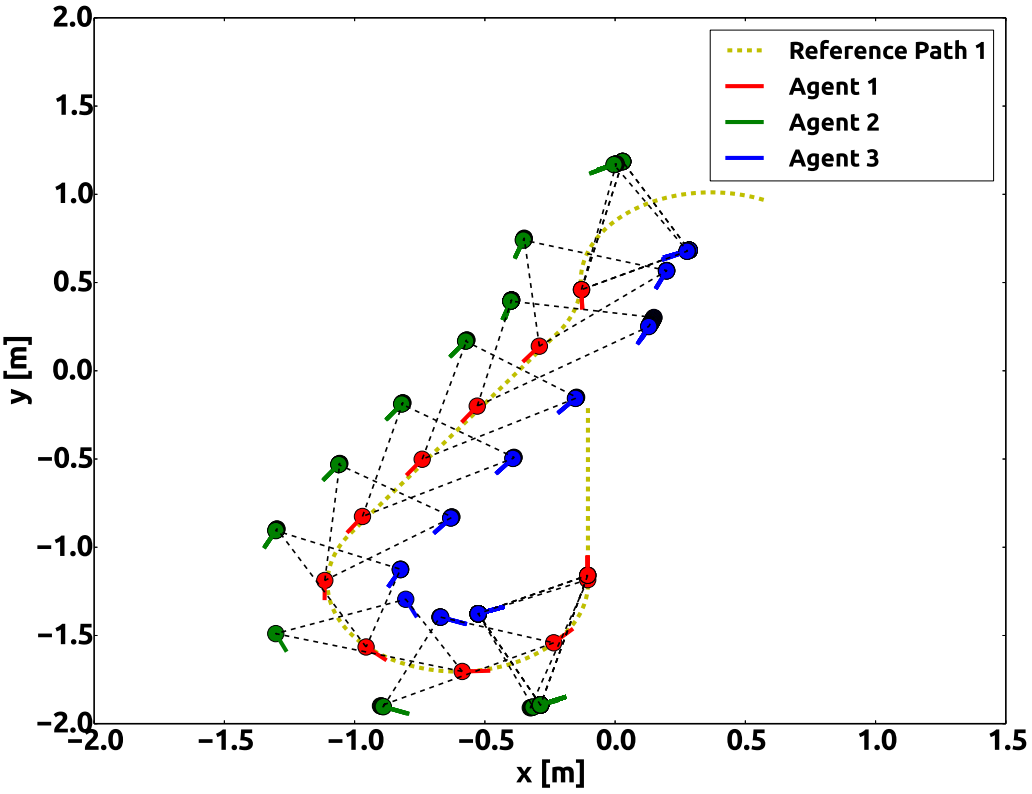
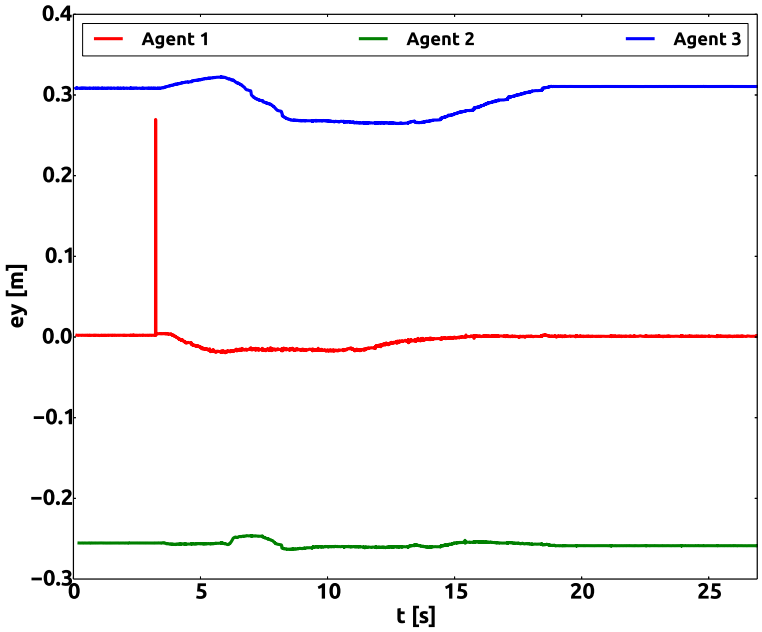
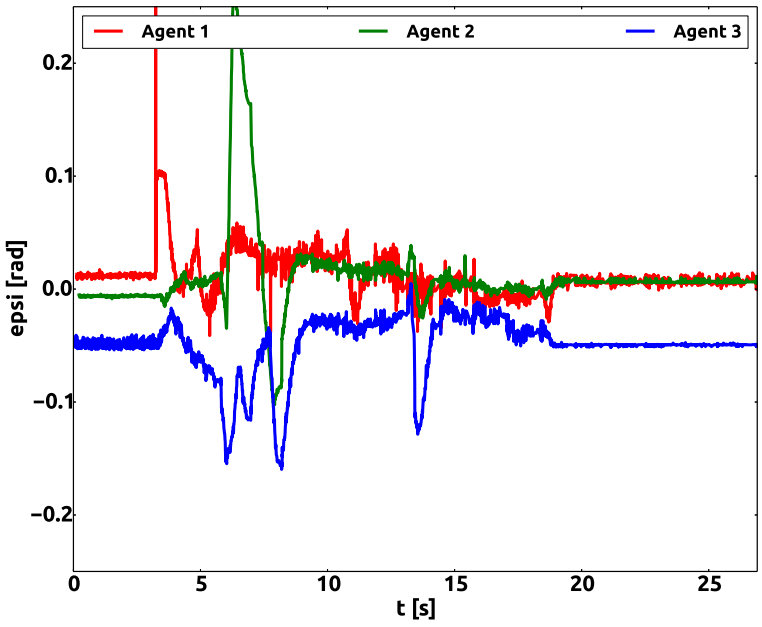


Figure 4-5: Movement of the formation in the x-y plane

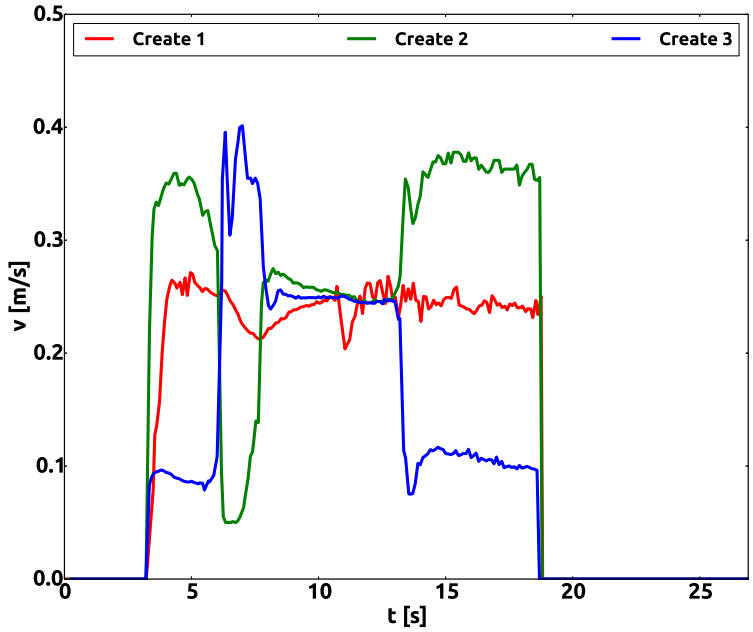


(a) Path center deviation e_y

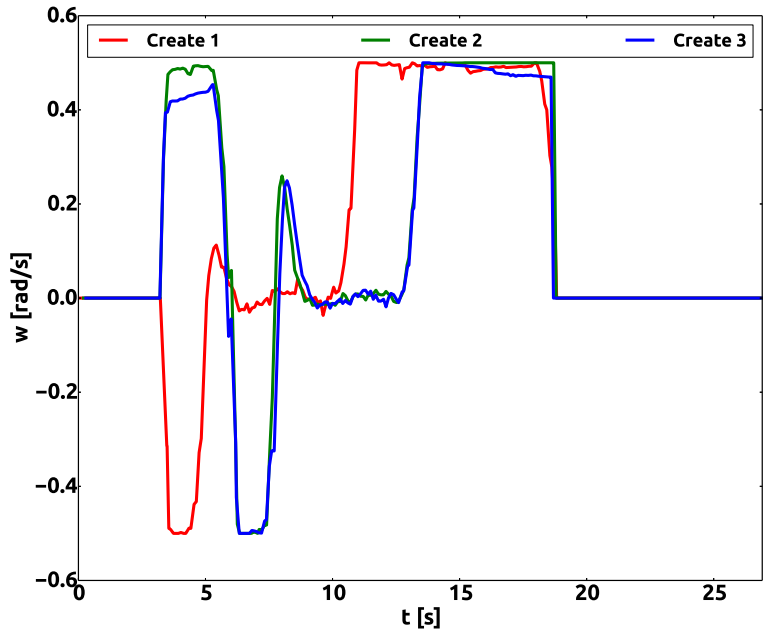


(b) Heading angle deviation e_ψ

Figure 4-6: Path tracking offsets



(a) Linear velocities v



(b) Angular velocities w

Figure 4-7: Agent velocities

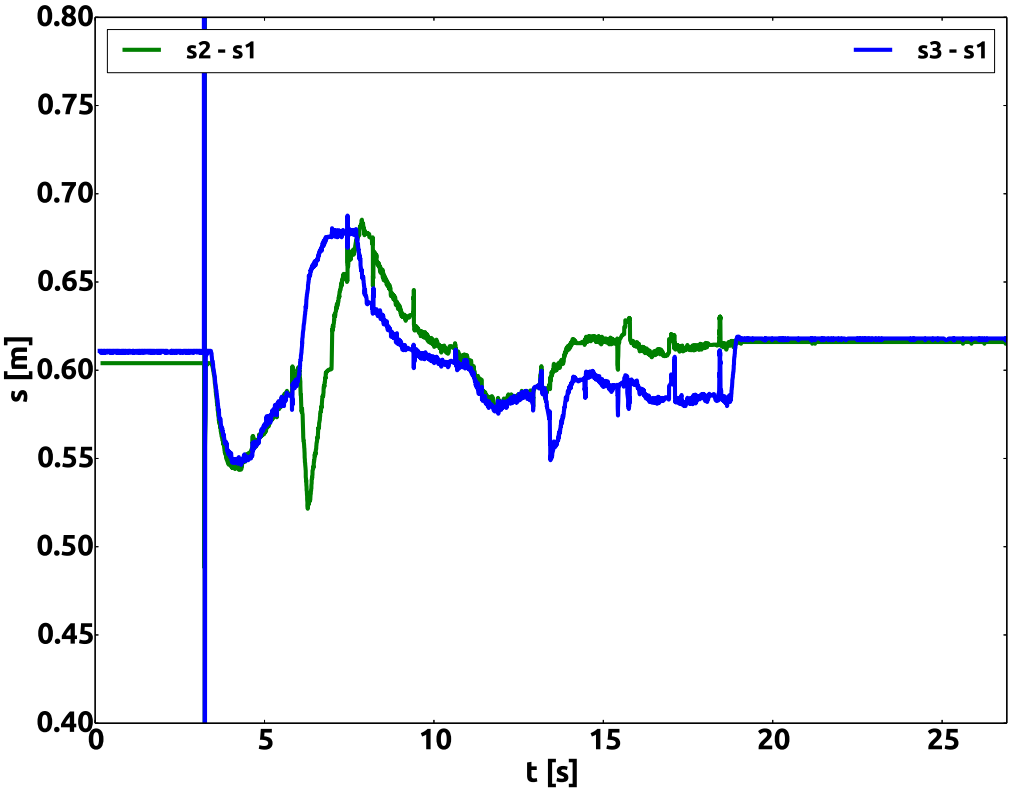


Figure 4-8: Formation keeping errors with respect to front agent

4-2-2 Agent failure

When an agent in the formation fails, the consensus algorithm will try to update the path reference velocity for each connected vehicle such that is slows down, waiting for the vehicle. The same triangular formation as before is considered here, but a certain point it is stopped. Figure 4-9 shows that agent 3 is stopped for a short period halfway the path.

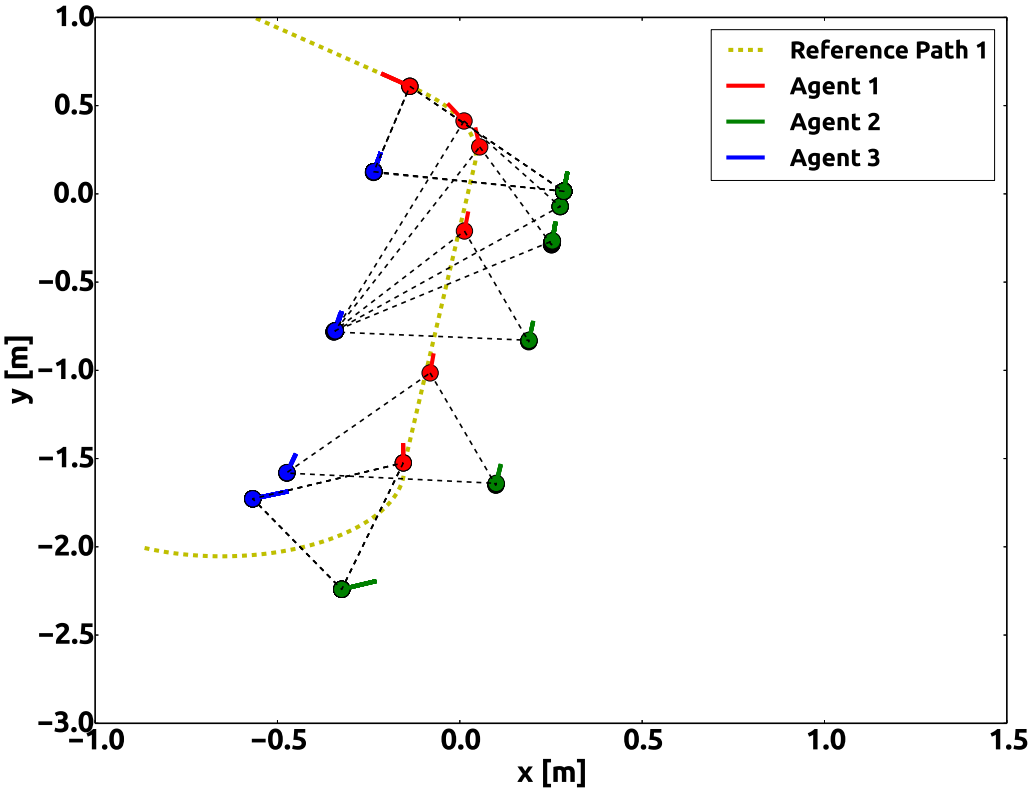
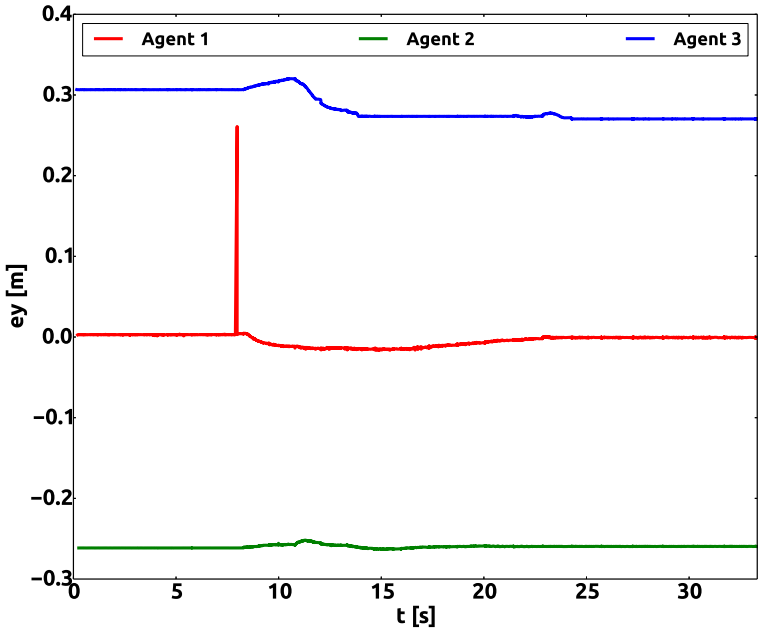
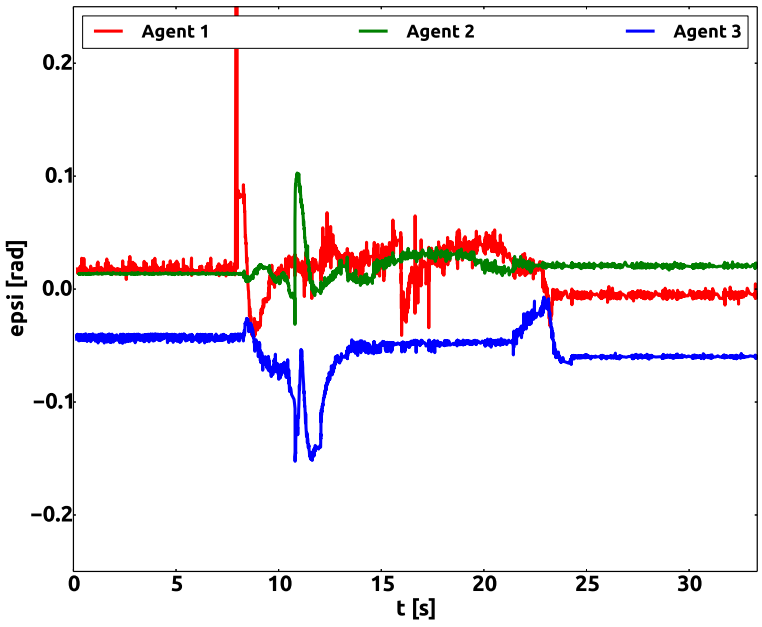


Figure 4-9: Movement of the formation in the x-y plane

At $t = 14$ the effect of stopping agent 3 can be seen in the calculated linear input velocities in Figure 4-11a. As soon the vehicle stops, the controller tries to accelerate it in an attempt to keep up with the other agents. The other agents can be seen to slow down, waiting for the other vehicle. At $t = 22$ the agent is allowed to continue and it quickly rejoins the formation.

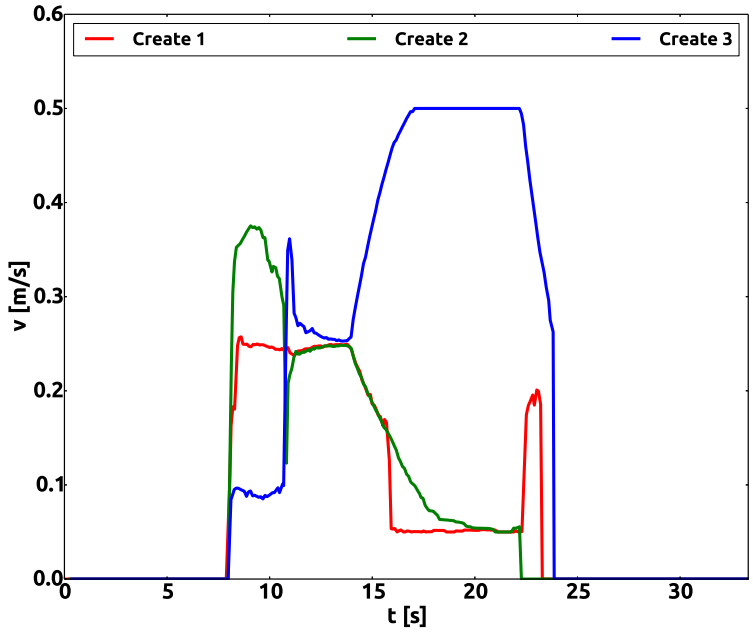


(a) Path center deviation e_y

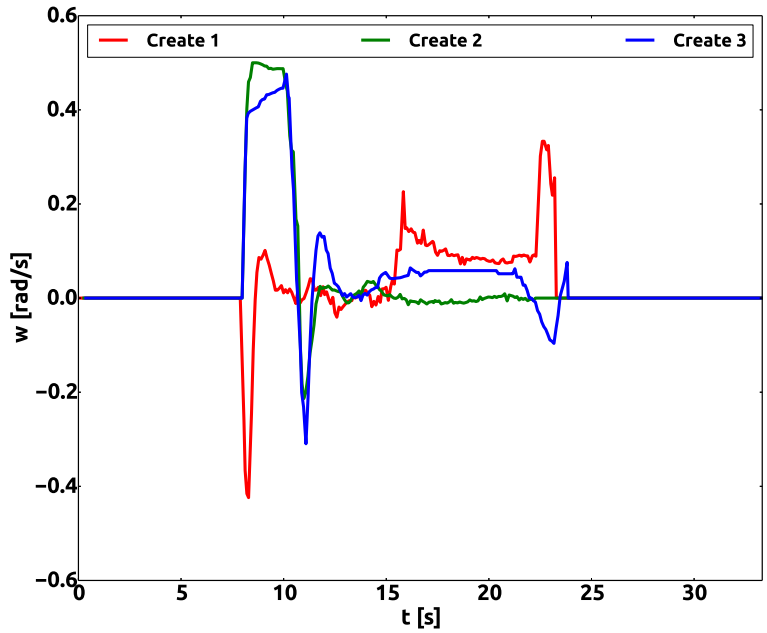


(b) Heading angle deviation e_ψ

Figure 4-10: Path tracking offsets



(a) Linear velocities v



(b) Angular velocities w

Figure 4-11: Agent velocities

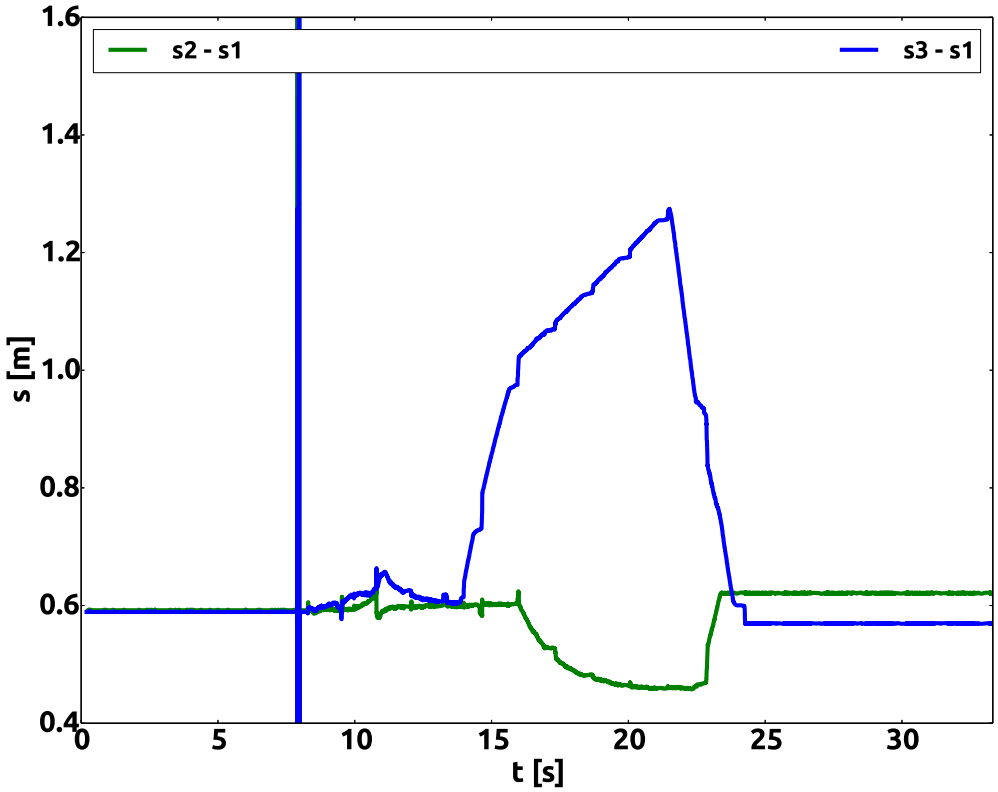


Figure 4-12: Formation keeping errors with respect to front agent

4-2-3 Evaluation

The Spatial Predictive Formation Controller converges the group agents to the desired formation in both simulation and practical experiment, even when confronted with vehicle failure. Within the formations, however, as can been seen in Figures 4-5 and 4-9 offsets, within bounds, are present due to noise and modeling errors. These errors are present due to several reason:

- **Controller tuning**: As this thesis work focused on experimental validation of the Spatial Predictive Control method as a formation controller, no specific performance goals were formulated. To this end, the controller has not been tuned towards optimal performance and can still be improved. This implies adjusting the prediction horizon, spatial discretization step size, and state weights. Choosing the right weight can drastically improve performance.
- **Modeling errors**: The ideal unicycle model is a simple kinematic model and assumes frictionless motion and direct achievement of input velocities. Although the Creates behave very well according to this ideal unicycle model, unmodeled dynamics remain. These dynamics differ for each agent due to the level of wear of its components. Propagating this modeling error through the spatial predictive controller increases the error.

- **Timing issues**: Each controller and model used in the simulation and experiment runs as a separate process and processes are therefore not publishing their information synchronously. This implies the time between calculating and executing a control input can come close to twice controllers sampling time of 10 Hz. On top of that, latency, although measured, has not been implemented in the current controller which can also increase errors.
- **Camera noise**: Although the OptiTrack cameras have millimeter precision, in some areas the tracking performance can deteriorate due to infrared noise and insufficient camera coverage. This will result in jitter in the reported agent pose which will propagate into the controller as well as consensus algorithm. This problem is well visualised in the agent heading angle errors in Figure 4-6b and 4-10b.

Chapter 5

Conclusions

The goal of this thesis was to develop a robotics lab capable of state-of-the-art robotics experiments. As a validation of the lab, a novel distributed formation control algorithm was developed implemented. This chapter provides a summary of the contributions that have been made and concludes with recommendations for future developments.

5-1 Summary on Spatial Predictive Formation Control

Spatial Predictive Control is a novel path tracking method that uses a conversion from time-dependent dynamics to spatial-dependent dynamics to generate path tracking inputs in a time-independent manner. These spatial dynamics express an agents position in terms of traveled path length and path offset. A Model Predictive Control structure is implemented to minimize path deviation whilst traveling along the path with a given reference velocity. To convert this method to a distributed formation control algorithm, consensus algorithms were introduced to analyze the flow of information between multiple agents in a distributed formation. Defining a formation keeping constraint between agents in terms of difference in traveled path length allowed for the choice of a simple consensus controller that updates the reference velocity along the path based on traveled path length only. The Model Predictive Controller using spatial-dependent dynamics is then used to track this reference velocity accurately. The resulting controller has been implemented and shown to work in both simulation and practice.

5-2 Summary on laboratory development

The development work performed on the laboratory resulted in a flexible and reliable setup with interesting features such as web based control. The hardware infrastructure of the Networked Embedded Robotics Lab features support for five different, three ground and two aerial, robotics platform using different connection protocols. For localization purposes when performing any experiment using these mobile platforms, a tracking system by Optitrack has

been installed and configured. All these components are connected through multiple PCs and a custom network. To provide students easy access to all of the hardware components in this laboratory, Robot Operating System (ROS) has been implemented. ROS is a state-of-the-art robotics libraries that provides the tools to separate complex robotics software into several smaller nodes. ROS's widespread acceptance and use imply that for virtually any robotics function, a software node is available. This allows students to easily use software nodes developed by other researchers or programmers and focus on the control algorithm they wish to develop. As an example of this, the implementation of the Spatial Predictive Formation controller was fully built in ROS on top of community built software nodes that connected to localization system and provided drivers for the used robotics platforms.

5-3 Recommendations on future developments

- An initial start has been made with the implementation of all robotic platforms available in the lab on ROS. Though fully functional for the Create and Sphero, work still has to be done on integration of the Parrot AR, hovercraft and Matrice DJI in the default ROS workflow. On top of this, more code examples can be developed to accommodate students in getting started in the lab.

- Within the formation of the optimal control problem the reference path tracking velocity is static at each evaluation and updated outside of the control loop through a consensus step. This means that the distance between agents, and therefore required reference keeping velocity, would remain fixed while moving along the path, or at least within the evaluation of the optimal control problem. Assuming agents in the formation will always make choose the optimal reference velocity, the position of other agents on the graph can be updated within the optimal control problem thereby creating a better estimate of the required reference keeping velocity at different positions along the path. This can create smoother velocity profiles and minimize chatter due to the consensus method. An example of this method is already given in the experiment description but not yet implemented.

- Currently a kinematic unicycle model is employed for the iCreates but this does not perfectly describe their dynamics. This effect is worsened by wear of the platforms and is different for each iCreate. As a model predictive control structure is used, a good model is a necessity. To achieve better results, a dynamic model should be implemented and identified for each iCreate. This model could be implemented by either changing the spatial predictive algorithm or could be implemented as a separate low-level controller that uses the Spatial Predictive Controller as a high-level planner.

- The developed controller was not tuned for optimal performance as no specific performance criteria were set. This was by choice, as the implementation of the spatial predictive formation controller was meant for demonstration purposes. Performance of the controller can be improved by further tuning the weights on the different control states.

- The velocity updating consensus approach is used outside of the Spatial Predictive Controller implying that the calculated reference velocity remains constant during the

entire control horizon. The consensus equation can be implemented within the Spatial Predictive Controller by making an assumption on how connected agents will update their velocity. This will likely result in a smoother calculated control input as the change in reference velocity while for formation keeping is reflected in the predictive control step.

Appendix A

Overview of Robotic Middleware

Table A-1: Overview of Robotic Middleware Part 1

Features	Orocos	MRPT	URBI	CLARAty	YARP
Simulation Environment	No	Yes	No	Yes	No
Architecture	C++ libraries	C++ libraries	Scripting language	Decision and function layer	Observer design pattern
Standards	ACE, TAO, CORBA	None	None	ACE, TAO, TCP, UDP	TCP, UDP
Distributed	No	No	Yes	Yes	Yes
Security	No	No	No	Yes	No
Fault tolerance	No	No	No	Yes	No
Real-time	Yes	Some libraries	Yes	Yes	No
Dynamic wiring	Yes	No	Yes	Partially	Yes
Open Source	LGPL	BSD	GPL and Commercial	Partially	LGPL
Platforms	Linux + Windows	Linux + Windows	Linux + Windows	Linux	Linux + Windows

Table A-2: Overview of Robotic Middleware Part 2

Features	MRDS	MOOS	Player	ROS
Simulation Environment	Yes	Yes	Stage (2D), Gazebo (3D)	Rviz
Architecture	Component-based, REST	Publisher / Subscriber	Client / Server	Publisher / Subscriber
Standards	.NET, SOA	TCP	TCP, UDP	RPC
Distributed	Yes	Yes	Yes	Yes
Security	Yes	No	Yes	No
Fault tolerance	No	No	No	Not explicit
Real-time	No	No	No	Yes
Dynamic wiring	Yes	Yes	Yes	Yes
Open Source	No	(L)GPL	GPL	BSD
Platforms	Windows	Linux & Windows	Linux & Windows	Linux

Table A-3: Overview of Robotic Middleware Part 3

Features	CARMEN	Webots	Rock	Fawkes
Simulation Environment	2D Simulator	Yes	Yes, MARS	Not included, Gazebo can be used
Architecture	3T hybrid architecture	Multiprocess architecture	Component based	Component based
Standards	TCP, IPC	Open Dynamics Engine	Uses shared memory	Uses shared memory
Distributed	Yes	No	No	Yes
Security	Yes	No	No	No
Fault tolerance	Yes	No	Yes	No
Real-time	No	Yes	Yes	Yes
Dynamic wiring	Yes	Yes	Yes	Yes
Open Source	GPL	No, commercial	GPL	GPL
Platforms	Linux	Linux & Windows	Linux & Windows	Linux

Table A-4: Properties of robotic middleware [8]

Properties	Description
Simulation environments	Testing software before implementation on an actual robot can speed up development considerably and prevent damage in experiments. Whether or not a framework supports a simulation environment can be an important factor, especially if expensive hardware is used. Another plus would be if the simulation environment can also be used for user input on actual platforms allowing combinations of virtual and real robots.
Architectural approaches	Defines the used programming paradigms. Examples are Client/Server or Publisher/Subscriber models to separate software in components.
Communication standards	Several protocols are available for platform independent data exchange, e.g. CORBA, ICE, ACE and ODE. These protocols can be important when interfacing other software or when certain software restrictions are available.
Distributed environment	This property defines whether software modules run and exchange data on different machines, allowing for a distributed implementation.
Security	In a controlled lab environment, security usually is not an issue. When deploying robots in real applications however or when remote access is required, communications should be safe. An example could be the support of SSL. For the distributed robotics lab, this is not a requirement but it is a plus.
Fault tolerance	Using a framework in real situations requires a framework to detect and recover from failures. Some frameworks explicitly incorporate these features whilst others inherit them from their distributed design.
Real-time support	Often in robotics real-time support is needed in order to guarantee safety or stability.
Dynamic wiring	Dynamic wiring allows connecting software components at runtime. If this is supported the flow of data can be changed as desired, greatly increasing usability.

Appendix B

Code

B-1 Unicycle model

```
#!/usr/bin/env python
import rospy
import tf

from numpy.matlib import *
from scipy.optimize import *

from visualization_msgs.msg import Marker
from geometry_msgs.msg import Twist, Pose2D


class Model:

    def __init__(self):

        self.x = 0
        self.y = 0
        self.psi = 0
        self.v = 0
        self.w = 0

        #Create rosnode
        rospy.init_node('create_spatial_sim')

        self.rate = rospy.Rate(20)
        self.dt = 1/20.

        #Listen to velocity input
        self.sub_vel = rospy.Subscriber('cmd_vel', Twist, self.listen)
        #Publish odometry data
```

```
31          self.pub_pose = rospy.Publisher('ground_pose', Pose2D, queue_size
                = 1)
32          self.pub_marker = rospy.Publisher('marker', Marker, queue_size =
                0)
33
34          #Start control loop
35          rospy.loginfo("Spatial simulation initialized.")
36          self.move()
37
38
39      def move(self):
40
41          msg = Pose2D()
42
43          while not rospy.is_shutdown():
44
45              dxdt = (self.v) * cos(self.psi)
46              dydt = (self.v) * sin(self.psi)
47              dpsidt = self.w
48
49              self.x = self.x + dxdt * self.dt
50              self.y = self.y + dydt * self.dt
51              self.psi = self.psi + dpsidt * self.dt
52              self.psi = (self.psi + pi) % (2 * pi) - pi
53
54              msg.x = self.x
55              msg.y = self.y
56              msg.theta = self.psi
57
58              # Publish odometry message
59              self.pub_pose.publish(msg)
60
61              #print "%.2f %.2f %.2f" % (self.x,self.y,self.psi)
62              ellipse = Marker()
63              ellipse.header.frame_id = "odom"
64              ellipse.header.stamp = rospy.Time.now()
65              ellipse.type = Marker.CYLINDER
66              ellipse.pose.position.x = self.x
67              ellipse.pose.position.y = self.y
68              ellipse.pose.position.z = 0
69              ellipse.pose.orientation.x = 0;
70              ellipse.pose.orientation.y = 0;
71              ellipse.pose.orientation.z = 0;
72              ellipse.pose.orientation.w = 1;
73              ellipse.scale.x = .2
74              ellipse.scale.y = .2
75              ellipse.scale.z = .1
76              ellipse.color.a = 1.0
77              ellipse.color.r = 1.0
78              ellipse.color.g = 1.0
79              ellipse.color.b = 1.0
80
81              # Publish the MarkerArray
```

```
82              self.pub_marker.publish(ellipse)
83
84
85              ##################
86              self.rate.sleep()
87
88      def listen(self,twist):
89          self.v = twist.linear.x
90          self.w = twist.angular.z
91
92
93  if __name__ == '__main__':
94      try:
95          m = Model()
96      except rospy.ROSInterruptException: pass
```

B-2 Objective function implementation

```
1   #!/usr/bin/env python
2   """
3   Created on Fri Apr  1 08:59:50 2016
4
5   @author: Frank de Winkel
6   """
7
8   #Import numpy and scipy within their own namespace
9   import numpy as np
10  import scipy.optimize as sp
11  #from path import ExtendedDubins
12
13  class SpatialController:
14
15      def __init__(self,Nc=5,Np=5,ds=0.05,dsdt_ref=0.2,Wey=100,Wepsi=5,
        Wdsdt=2,Wdu=1):
16          """Initialization of Spatial Predictive Controller class
17          Parameters
18          ----------
19          Nc : int (default = 5)
20               Control horizon
21          Np : int (default = 5)
22               Prediction horizon, cannot be smaller than control horizon
23          ds : float (default = 0.1)
24               Spatial step size
25          dsdt_ref : float (default = 0.25)
26               Reference velocity along path
27          Wey : float (default = 10)
28               Cost function weight on path deviation
29          Wepsi : float (default = 10)
30               Cost function weight on heading angle deviation
31          Wdsdt : float (default = 1)
32               Cost function weight on reference speed deviation
33          Wdu : float (default = 1)
```

```
34              Cost function weight on input change
35          """
36
37          self.Nc = max(1,int(Nc))            #Set control horizon
38          self.Np = max(1,int(Nc),int(Np))    #Set prediction horizon
39
40          #Construct bounds
41          self.v_min = 1E-2
42          self.v_max = .5
43          self.w_min = -.5
44          self.w_max = .5
45          self.bounds = []
46          for i in range(0,self.Nc):
47              self.bounds.insert(0,(self.v_min,self.v_max))
48              self.bounds.append((self.w_min,self.w_max))
49          self.bounds = tuple(self.bounds)
50
51          #Desired offset
52          self.dey = 0
53
54          #Create u0
55          self.u0 = np.concatenate((np.ones(self.Nc)*dsdt_ref, np.zeros(
                self.Nc)))
56          self.umin = np.concatenate((np.ones(self.Nc)*self.v_min, np.ones(
                self.Nc)*self.w_min))
57          self.umax = np.concatenate((np.ones(self.Nc)*self.v_max, np.ones(
                self.Nc)*self.v_max))
58
59          #Float inputs are divided by 1. to ensure that they are floats
60          self.ds = ds / 1.                   #Spatial discretization step
61          self.dsdt_ref = dsdt_ref / 1.       #Reference path velocity
62          self.Wey = Wey / 1.                 #Cost function weights
63          self.Wepsi = Wepsi / 1.
64          self.Wdsdt = Wdsdt / 1.
65          self.Wdu = Wdu / 1.
66
67      def cost(self,u,dpsi,ey,epsi,vt,wt):
68          """Spatial predictive cost function
69          Parameters
70          ----------
71          u : array of size (1 x 2*Nc)
72              linear and angular velocity vector
73          dpsi : array (1 x Np)
74              road heading angle difference vector
75          ey : float
76              path offset
77          epsi : float
78              heading angle error
79          Returns
80          -------
81          J : float
82              The total cost is returned based on the supplied input vector
83          """
```

```
84
85          v = u[0:self.Nc]              #Retrieve linear velocity from input
                vector
86          vd = np.diff(np.insert(v,0,vt))             #Change in linear
                velocity
87          w = u[self.Nc:2*self.Nc]     #Retrieve angular velocity from input
                 vector
88          wd = np.diff(np.insert(w,0,wt))             #Change in angular
                velocity
89
90          J = self.Wdu * (vd.T.dot(vd) + wd.T.dot(wd)) #Initial value of
                cost function
91
92          #Loop over prediction horizon
93          for i in range(0,self.Np):
94
95              vi = v[min(self.Nc-1,i)]
96              wi = w[min(self.Nc-1,i)]
97
98              dpsids = dpsi[i]
99              if dpsids == 0:
100                 ps = 9E9
101             else:
102                 ps = 1. / dpsids
103
104             #Get advancement along path w.r.t time
105             dsdt = 1 / (1 - ey / ps) * vi * np.cos(epsi)
106
107             #Advancement of angle and position error
108             deyds = vi * np.sin(epsi) / dsdt
109             depsids = wi / dsdt - dpsids
110
111             #Update error
112             ey = ey + self.ds * deyds
113             epsi = (epsi + self.ds * depsids + np.pi) % (2 * np.pi) - np.
                    pi
114
115             #Update cost
116             J = J + self.Wey * (self.dey-ey)**2 + self.Wepsi * epsi**2 +
                    self.Wdsdt * (dsdt-self.dsdt_ref)**2
117
118         return J
119
120     def minimize(self,dpsi,ey,epsi,v,w):
121         """Calculate optimal control action
122         Parameters
123         ----------
124         dpsi : array of size (1 x Np)
125             Vehicle reference heading angle change
126         Returns
127         -------
128         u : tuple
129             Returns tuple of linear and angular velocity (v, w)
```

```
130         """
131
132         #Run the minimization algorithm
133         result = sp.minimize(self.cost, self.u0, args=(dpsi,ey,epsi,v,w),
                bounds=self.bounds, method="SLSQP")
134
135         for i in range(0,self.Nc):
136             self.u0[i]          = min(self.v_max, max(self.v_min, result.
                    x[i]))
137             self.u0[self.Nc+i]  = min(self.w_max, max(self.w_min, result.
                    x[self.Nc+i]))
138
139         #Return optimal input
140         return self.u0[0], self.u0[self.Nc]
```

B-3 Path planner

```
1  #!/usr/bin/env python
2  """
3  Created on Fri Apr  1 08:59:50 2016
4
5  @author: Frank de Winkel
6  """
7
8  #Import numpy and scipy within their own namespace
9  import numpy as np
10 import scipy.optimize as sp
11 #from path import ExtendedDubins
12
13 class SpatialController:
14
15     def __init__(self,Nc=5,Np=5,ds=0.05,dsdt_ref=0.2,Wey=100,Wepsi=5,
           Wdsdt=2,Wdu=1):
16         """Initialization of Spatial Predictive Controller class
17         Parameters
18         ----------
19         Nc : int (default = 5)
20              Control horizon
21         Np : int (default = 5)
22              Prediction horizon, cannot be smaller than control horizon
23         ds : float (default = 0.1)
24              Spatial step size
25         dsdt_ref : float (default = 0.25)
26              Reference velocity along path
27         Wey : float (default = 10)
28              Cost function weight on path deviation
29         Wepsi : float (default = 10)
30              Cost function weight on heading angle deviation
31         Wdsdt : float (default = 1)
32              Cost function weight on reference speed deviation
33         Wdu : float (default = 1)
34              Cost function weight on input change
```

```
35          """
36
37          self.Nc = max(1,int(Nc))            #Set control horizon
38          self.Np = max(1,int(Nc),int(Np))    #Set prediction horizon
39
40          #Construct bounds
41          self.v_min = 1E-2
42          self.v_max = .5
43          self.w_min = -.5
44          self.w_max = .5
45          self.bounds = []
46          for i in range(0,self.Nc):
47              self.bounds.insert(0,(self.v_min,self.v_max))
48              self.bounds.append((self.w_min,self.w_max))
49          self.bounds = tuple(self.bounds)
50
51          #Desired offset
52          self.dey = 0
53
54          #Create u0
55          self.u0 = np.concatenate((np.ones(self.Nc)*dsdt_ref, np.zeros(
                self.Nc)))
56          self.umin = np.concatenate((np.ones(self.Nc)*self.v_min, np.ones(
                self.Nc)*self.w_min))
57          self.umax = np.concatenate((np.ones(self.Nc)*self.v_max, np.ones(
                self.Nc)*self.v_max))
58
59          #Float inputs are divided by 1. to ensure that they are floats
60          self.ds = ds / 1.                   #Spatial discretization step
61          self.dsdt_ref = dsdt_ref / 1.       #Reference path velocity
62          self.Wey = Wey / 1.                 #Cost function weights
63          self.Wepsi = Wepsi / 1.
64          self.Wdsdt = Wdsdt / 1.
65          self.Wdu = Wdu / 1.
66
67      def cost(self,u,dpsi,ey,epsi,vt,wt):
68          """Spatial predictive cost function
69          Parameters
70          ----------
71          u : array of size (1 x 2*Nc)
72              linear and angular velocity vector
73          dpsi : array (1 x Np)
74              road heading angle difference vector
75          ey : float
76              path offset
77          epsi : float
78              heading angle error
79          Returns
80          -------
81          J : float
82              The total cost is returned based on the supplied input vector
83          """
84
```

```
85          v = u[0:self.Nc]            #Retrieve linear velocity from input
                vector
86          vd = np.diff(np.insert(v,0,vt))             #Change in linear
                velocity
87          w = u[self.Nc:2*self.Nc]    #Retrieve angular velocity from input
                 vector
88          wd = np.diff(np.insert(w,0,wt))             #Change in angular
                velocity
89
90          J = self.Wdu * (vd.T.dot(vd) + wd.T.dot(wd)) #Initial value of
                cost function
91
92          #Loop over prediction horizon
93          for i in range(0,self.Np):
94
95              vi = v[min(self.Nc-1,i)]
96              wi = w[min(self.Nc-1,i)]
97
98              dpsids = dpsi[i]
99              if dpsids == 0:
100                 ps = 9E9
101             else:
102                 ps = 1. / dpsids
103
104             #Get advancement along path w.r.t time
105             dsdt = 1 / (1 - ey / ps) * vi * np.cos(epsi)
106
107             #Advancement of angle and position error
108             deyds = vi * np.sin(epsi) / dsdt
109             depsids = wi / dsdt - dpsids
110
111             #Update error
112             ey = ey + self.ds * deyds
113             epsi = (epsi + self.ds * depsids + np.pi) % (2 * np.pi) - np.
                    pi
114
115             #Update cost
116             J = J + self.Wey * (self.dey-ey)**2 + self.Wepsi * epsi**2 +
                    self.Wdsdt * (dsdt-self.dsdt_ref)**2
117
118         return J
119
120     def minimize(self,dpsi,ey,epsi,v,w):
121         """Calculate optimal control action
122         Parameters
123         ----------
124         dpsi : array of size (1 x Np)
125             Vehicle reference heading angle change
126         Returns
127         -------
128         u : tuple
129             Returns tuple of linear and angular velocity (v, w)
130         """
```

```
131
132         #Run the minimization algorithm
133         result = sp.minimize(self.cost, self.u0, args=(dpsi,ey,epsi,v,w),
                bounds=self.bounds, method="SLSQP")
134
135         for i in range(0,self.Nc):
136             self.u0[i]          = min(self.v_max, max(self.v_min, result.
                    x[i]))
137             self.u0[self.Nc+i]  = min(self.w_max, max(self.w_min, result.
                    x[self.Nc+i]))
138
139         #Return optimal input
140         return self.u0[0], self.u0[self.Nc]
```

B-4 Spatial conversion

```
1  #!/usr/bin/env python
2  """
3  Created on Wed Mar 30 12:23:23 2016
4
5  @author: Frank de Winkel
6  """
7
8  from numpy.matlib import *
9
10 class SpatialConversion():
11
12     def __init__(self, ds = 1E-3):
13         self.path = None
14         self.ds = ds / 1.
15         self.sf = 0
16
17     def setPath(self, path):
18         self.path = path
19
20     def getPath(self, s0, s1, ds):
21         """Get path information between two points
22         Parameters
23         ----------
24         xs : array (1 x 3)
25              Initial pose
26         xf : array (1 x 3)
27              Final pose
28         Returns
29         -------
30         dpsi : array ( 1 x (s1-s0)/ds )
31         """
32
33         #result
34         res = []
35         #Derivative of angle error is given by
36         dpsids = diff(self.path[:,2]) / self.ds
```

```
37          #max-index
38          imax = len(dpsids) - 1
39          #Create subsvector for control loop
40          for s in arange(s0, s1, ds):
41              res.append( dpsids[ min( s / self.ds , imax ) ] )
42          return res
43
44      def getState(self, x):
45          """Convert pose to spatial pose
46          Parameters
47          ----------
48          x : array (1 x 3)
49               Robot pose
50          Returns
51          -------
52          tuple : s, ey, epsi
53          """
54
55          #Find the index of the closest pose
56          dist_2 = sum((self.path[:,0:2] - x[0:2])**2, axis=1)
57          index = argmin(dist_2)
58
59          #Distance traveled along the path is given by index times
                discretization size
60          s = self.ds * index
61
62          #Get the closest pose
63          p = self.path[index]
64
65          #Determine deviation (check this code!!!!)
66          error = x[0:2] - p[0:2]
67          angle = -(p[2]+pi/2)
68          R = array([ [ cos(angle) , -sin(angle) ], [ sin(angle) , -cos(
                angle) ] ])
69          error = R.dot(error)
70          ey = error[0]
71
72          #angle error
73          epsi = (x[2] - p[2] + pi) % (2 * pi) - pi
74
75          #Return result as tuple
76          return s, ey, epsi
```

B-5 Spatial controller

```
1  #!/usr/bin/env python
2  """
3  Created on Mon Apr  4 12:48:06 2016
4
5  @author: dcscm
6  """
7
```

```
8  import rospy
9  import numpy as np
10
11 from geometry_msgs.msg import Twist
12 from geometry_msgs.msg import Pose2D
13
14 from spc_conversion import SpatialConversion
15 from spc_dubins import SpatialController
16
17 from spatial.msg import SpatialPose
18 from spatial.msg import Path2D
19
20 class Controller:
21
22     def __init__(self):
23
24         rospy.init_node('create_spatial_controller')
25
26         self.conv = SpatialConversion()
27         self.SPC = SpatialController()
28
29         self.s = 0
30         self.ey = 0
31         self.epsi = 0
32         self.x = np.array([0,0,0])
33         self.v = 0
34         self.w = 0
35
36         self.initialized = False
37
38         self.pub_velocity = rospy.Publisher('cmd_vel', Twist, queue_size
               =1)
39         self.pub_spatial_state = rospy.Publisher('spatial_pose',
               SpatialPose, queue_size = 1)
40         self.sub_spatial_path = rospy.Subscriber('/spatial/path', Path2D,
                self.setPath)
41         self.sub_state = rospy.Subscriber('ground_pose', Pose2D, self.
               setPose)
42
43         self.ds = rospy.get_param('~ds',0)
44         self.SPC.dey = rospy.get_param('~dey',0)
45         self.connected_to = rospy.get_param('~connected_to',[])
46         self.subs = []
47         self.states = []
48         for node in self.connected_to:
49             self.states.append([0,node[1]])
50             self.subs.append(rospy.Subscriber('/create'+str(node[0])+'/
                   spatial_pose',SpatialPose,self.getLeaders,callback_args=(
                   len(self.states)-1)))
51             rospy.loginfo("Subscribed to /create"+str(node[0])+'/
                   spatial_pose')
52
53         self.rate = rospy.Rate(10)
```

```
54
55          self.track()
56
57      def setPath(self, path):
58          points = []
59          for p in path.poses:
60              points.append((p.x,p.y,p.theta))
61          self.conv.setPath(np.asarray(points))
62          self.conv.sf = path.length
63          self.initialized = True
64
65      def getLeaders(self, pose, index):
66          self.states[index][0] = pose.s
67
68      def setPose(self, pose):
69          #Get pose from topic
70          self.x = np.array([pose.x, pose.y, pose.theta])
71          #If path is available, convert to spatial pose
72          if self.initialized == True:
73              self.s, self.ey, self.epsi = self.conv.getState(self.x)
74              #Publish spatial pose
75              spatialPose = SpatialPose()
76              spatialPose.s = self.s
77              spatialPose.ey = self.ey
78              spatialPose.epsi = self.epsi
79              self.pub_spatial_state.publish(spatialPose)
80
81      def track(self):
82
83          twist = Twist()
84
85          while not rospy.is_shutdown():
86
87              if self.initialized == True:
88
89                  self.SPC.dsdt_ref = 0
90                  for state in self.states:
91                      self.SPC.dsdt_ref = self.SPC.dsdt_ref + 0.3 * ((state
                            [1] - self.ds) - (state[0] - self.s))
92                  self.SPC.dsdt_ref = max(0, min(1, 0.2 - self.SPC.dsdt_ref
                        / len(self.states) ))
93
94                  dpsi = self.conv.getPath(self.s, self.s+self.SPC.ds*self.
                        SPC.Np, self.SPC.ds)
95
96                  if self.s < self.conv.sf + self.ds + 0.5:
97                      v, w = self.SPC.minimize(dpsi, self.ey, self.epsi,
                            self.v, self.w)
98                  elif abs(self.epsi) > 0.05:
99                      v, w = self.SPC.minimize(dpsi, self.ey, self.epsi,
                            self.v, self.w)
100                     v = 0
101                 else:
```

```
102                     v, w = 0, 0
103
104                 self.v = v
105                 self.w = w
106
107                 twist.linear.x = v
108                 twist.angular.z = w
109
110                 self.pub_velocity.publish(twist)
111
112                 #rospy.loginfo( "%.3f , %.3f " % (self.ey, self.epsi))
113                 #rospy.loginfo(self.states)
114
115                 self.rate.sleep()
116
117 if __name__ == '__main__':
118     try:
119         c = Controller()
120     except rospy.ROSInterruptException: pass
```

Bibliography

[1] Y. Goa, A. Gray, J. V. Frasch, L. Theresa, E. Tseng, J. K. Hedrick, and F. Borrelli, "Spatial Predictive Control for Agile Semi-Autonomous Ground Vehicles," *Proceedings of the 11th International Symposium on Advanced Vehicle Control*, 2012.

[2] S. LaValle, "Path generation." http://planning.cs.uiuc.edu/img6687.gif, 2017. [Online; accessed Januari 25, 2017].

[3] Sphero, "Orbotix." https://images-na.ssl-images-amazon.com/images/I/51h6bMLVNKL._SL1136_.jpg, 2017. [Online; accessed Jan 25, 2017].

[4] Parrot, "Parrot ar drone." https://www.quadrocoptershop.nl/content/images/thumbs/0000019_parrot-ardrone-20-geel.jpeg, 2016. [Online; accessed May 9, 2016].

[5] DJI, "Matrice dji 100." http://asset1.djicdn.com/images/360/matrice100/platform/0.png, 2017. [Online; accessed Januari 25, 2017].

[6] OptiTrack, "Optihub." http://www.optitrack.com/products/optihub/, 2017. [Online; accessed Januari 25, 2017].

[7] OptiTrack, "Motive tracking software." https://www.optitrack.com/static/images/motiveTrackerScreenRigidBodySwarmCropped.png, 2017. [Online; accessed Januari 25, 2017].

[8] A. Elkady and T. Sobh, "Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography," *Journal of Robotics*, vol. 2012, pp. 1–15, 2012.

[9] W. Guanghua, L. Deyi, G. Wenyan, and J. Peng, "Study on formation control of multi-robot systems," in *Third International Conference on Intelligent System Design and Engineering Applications*, pp. 1335–1339, 2013.

[10] A. Fujimori, T. Fujimoto, and G. Bohacs, "Distributed Leader-Follower Navigation of Mobile Robots," in *International Conference on Control and Automation (ICCA)*, vol. 2, pp. 960–965, 2005.

[11] K. Kanjanawanishkul, X. Li, and A. Zell, "Nonlinear model predictive control of omnidirectional mobile robot formations," in *Second International Conference on Robot Communication and Coordination. ROBOCOMM '09.*, May 2009.

[12] K.-H. Tan and M. A. Lewis, "High-Precision Formation Control of Mobile Robots using Virtual Structures," *Autonomous Robots*, vol. 4, no. 4, pp. 387–403, 1997.

[13] E. J. Gomez, F. Martinez Santa, and F. H. Martinez Sarmiento, "A Comparative Study of Geometric Path Planning Methods for a Mobile Robot : Potential Field and Voronoi Diagrams," in *II International Congress of Engineering Mechatronics and Automation (CIIMA)*, pp. 1–6, IEEE, 2013.

[14] R. Olfati-Saber and R. M. Murray, "Distributed cooperative control of multiple vehicle formations using structural potential functions," *IFAC World Congress*, pp. 346–352, 2002.

[15] M. Bartulovic, I. Palunko, and S. Bogdan, "Formation Control Using Adaptive Parameter-Dependent Potential Functions," in *IEEE Conference on Control Applications (CCA)*, pp. 530–535, IEEE, 2014.

[16] W. Ren, "Consensus Based Formation Control Strategies for Multi-vehicle Systems," in *Proceedings of the 2006 American Control Conference*, pp. 4237–4242, June 2006.

[17] W. Ren, "Multi-vehicle Consensus with a Time-varying Reference State," *Systems & Control Letters 56*, pp. 474–483, 2007.

[18] K. Kanjanawanishkul, "MPC-Based Path Following Control of an Omnidirectional Mobile Robot with Consideration of Robot Constraints," *Advances in Electrical and Electronic Engineering*, 2015.

[19] A. Siddaramappa, "Rigid formation control using hovercrafts," Master's thesis, TU Delft, Nov. 2015.

[20] M. P. Do Carmo, *Differential geometry of curves and surfaces*, vol. 2. Prentice-hall Englewood Cliffs, 1976.

[21] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[22] W. Garage, "ROS Website." http://www.ros.org/.

[23] D. Kraft, *A Software Package for Sequential Quadratic Programming*. Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt Köln: Forschungsbericht, Wiss. Berichtswesen d. DFVLR, 1988.

[24] S. Developers, "Scipy.org." https://www.scipy.org/, 2017. [Online; accessed Januari 25, 2017].

[25] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for Robotics: A Survey," in *IEEE Conference on Robotics, Automation and Mechatronics*, pp. 736–742, IEEE, September 2008.

[26] P. J. Conroy, "The Development of an Aerial Robotics Laboratory Highlighting the First Experimental Validation of Optimal Reciprocal Collision Avoidance," Master's thesis, University of Utah, Aug. 2013.

[27] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, "The Office Marathon: Robust Navigation in an Indoor Office Environment," in *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 300–307, IEEE, May 2010.

[28] M. A. Ma'sum, G. Jati, M. K. Arrofi, A. Wibowo, P. Mursanto, and W. Jatmiko, "Autonomous Quadcopter Swarm Robots For Object Localization and Tracking," in *International Symposium on Micro-NanoMachatronics and Human Science (MHS)*, pp. 1–6, IEEE, November 2013.

[29] KU Leuven, "OROCOS Website." http://www.orocos.org/.

[30] H. Bruyninckx, "Open Robot Control Software: the OROCOS Project," in *Proceedings of the 2001 IEEE International Conference on Robotics and Automation*, vol. 3, pp. 2523–2528, IEEE, May 2001.

[31] O. M. Group, "Corba Website." http://www.corba.org/.

[32] Gadeyne and Klaas, "BFL: Bayesian Filtering Library." http://www.orocos.org/bfl, 2001.

[33] R. Smits, T. De Laet, K. Claes, H. Bruyninckx, and J. De Schutter, "iTASC: a Tool for Multi-Sensor Integration in Robot Manipulation," in *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pp. 426–433, IEEE, August 2008.

[34] W. Fetter Lages, D. Ioris, and D. C. Santini, "An Architecture for Controlling the Barrett WAM Robot Using ROS and OROCOS," in *Proceedings of ISR/Robotik 2014; 41st International Symposium on Robotics*, pp. 649–656, VDE, 2014.

[35] K. Stoy, "Player Stage Website." http://playerstage.sourceforge.net/.

[36] M. Kranz, R. B. Rusu, A. Maldonado, M. Beetz, and A. Schmidt, "A Player/Stage System for Context-Aware Intelligent Environments," in *Proceedings of UbiSys'06, System Support for Ubiquitous Computing Workshop, at the Annual Conference on Ubiquitous Computing*, 2006.

[37] T. H. J. Collett, B. A. MacDonald, and B. P. Gerkey, "Player 2.0: Toward a Practical Robot Programming Framework," in *Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*, 2005.

[38] W. Garage, "Ros indigo." http://wiki.ros.org/indigo.

[39] iRobot, "iRobot Create." http://www.irobot.com/.

[40] V. Robotics, "The irobot create." http://verifiablerobotics.com/CreateMATLABsimulator/create.jpg, 2016. [Online; accessed May 9, 2016].

[41] MIT, "Cricket sensor." http://cricket.csail.mit.edu/pictures/cricketv2.jpg, 2016. [Online; accessed May 9, 2016].

[42] MIT, "Cricket Main Website." http://cricket.csail.mit.edu/.

[43] TurtleBot, "Turtlebot Robotic Platform." http://turtlebot.com/.

[44] T. R. Wiki, "Turtlebot Robotic Platform ROS Package." http://wiki.ros.org/turtlebot.

[45] S. Kandasamy, "Slamming with spheros: An impact-based approach to simultaneous localization and mapping," Master's thesis, TU Delft, Aug. 2015.

[46] M. Wise, "Sphero ROS Drivers." https://github.com/mmwise/sphero_ros.

[47] R. P. K. Jain, "Transportation of cable suspended load using unmanned aerial vehicles," Master's thesis, TU Delft, August 2015.

[48] OptiTrack, "OptiTrack Website." http://www.optitrack.com/.

[49] R. Curnow and M. Lichvar, "Chrony." https://chrony.tuxfamily.org/, 2017. [Online; accessed Jan 25, 2017].

[50] C. Crick, "Ros and rosbridge: Roboticists out of the loop," *7th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, 2012.

[51] F. Valentinis, "Path generation for the dubin's car." https://pypi.python.org/pypi/dubins, 2017. [Online; accessed Januari 25, 2017].

Glossary

List of Acronyms

DCSC	Delft Center for Systems and Control
ROS	Robot Operating System
NER	Networked Embedded Robotics Lab