UNIVERSITY OF ZAGREB
FACULTY OF MECHANICAL ENGINEERING AND
NAVAL ARCHITECTURE

MASTER'S THESIS

Matija Rossi

Zagreb, 2014

UNIVERSITY OF ZAGREB
FACULTY OF MECHANICAL ENGINEERING AND NAVAL ARCHITECTURE

MASTER'S THESIS

Supervisor:

Doc. dr. sc. Andrej Jokić

Student:

Matija Rossi

Zagreb, 2014

I declare that I have made this work independently, using the knowledge acquired during my studies and the cited literature.

I would like to thank my supervisor, Professor Andrej Jokić, for his continuous help and support, for all the time he dedicated to me, for all the knowledge I acquired from him, and without whom this work would not have been possible.

I also wish to thank my family, my girlfriend, and all of my friends for their support and understanding during my studies.

Matija Rossi

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za diplomske ispite studija strojarstva za smjerove:
proizvodno inženjerstvo, računalno inženjerstvo, industrijsko inženjerstvo i menadžment, inženjerstvo materijala i mehatronika i robotika

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa:	
Ur.broj:	

DIPLOMSKI ZADATAK

Student: **Matija Rossi** Mat. br.: 0035178935

Naslov rada na hrvatskom jeziku: **ROS programski paket za distribuirano upravljanje mrežama dinamičkih sustava**

Naslov rada na engleskom jeziku: **ROS package for distributed control of networks of dynamical systems**

Opis zadatka:
Examples of networks of dynamical systems include platoons of vehicles on automated motorways; groups of ground mobile robots, unmanned aerial vehicles or autonomous underwater vehicles which collaborate towards a common goal; intelligent structures with active vibration damping or fluid flow control; adaptive optics; smart power grids. Distributed control is a control structure in which each subsystem in a dynamical network is controlled locally, while every local controller can exchange information with a, usually small, subset of the remaining controllers.

Within this thesis it is necessary to design and develop a software package for flexible and efficient modelling and simulation of networks of dynamical systems with distributed control structures. The package needs to be implemented within the Robot Operating System (ROS) framework and it has to fulfil the following requirements:

- Offer the possibility of implementation (modelling and simulation) of distributed controllers with a control law given in explicit and closed form, with a single information exchange within one sampling period;
- Offer the possibility of implementation (modelling and simulation) of distributed controllers with iterative communication within one sampling period, e.g., control laws that solve a global (at the network level) optimisation problem on-line;
- Have a modular structure that allows to easily modify the number of subsystems in a network;
- Have a modular structure that allows straightforward implementation of controllers onto real physical systems, i.e. directly replacing a simulated plant with a physical system (e.g. a robot).

A short overview of the state-of-the-art control synthesis methods and approaches has to be given, as well as an overview of distributed control applications. It is also necessary to demonstrate the operation and usability of the developed software using simulations of realistic examples (e.g. control of a group of ground, aerial, or underwater autonomous vehicles; control of interconnected mechanical pendulums; and/or similar).

Zadatak zadan:	Rok predaje rada:	Predviđeni datum obrane:
8. svibnja 2014.	10. srpnja 2014.	16., 17. i 18. srpnja 2014.

Zadatak zadao:

[illegible]

Doc.dr.sc. Andrej Jokić

Predsjednik Povjerenstva:

[illegible]

Prof. dr. sc. Franjo Cajner

Abstract

In the field of distributed control of dynamical networks, the scientific community is currently focused on developing a general underlying theory for efficient control synthesis. However, a universal software framework for development, simulation, testing, and real-life implementation of control algorithms for such systems is still missing. This thesis explores the possibilities of developing such a tool, which would minimise the transition from simulation to implementation. A fully working framework has been developed and is now being presented, explaining in detail its architecture and usage. Several networks have been implemented inside it to prove its efficiency.

Keywords: dynamical networks; distributed control; decentralised control; software framework; ROS

Prošireni sažetak

Prateći brzi razvoj komunikacijskih i informacijskih tehnologija, the razvoj novih generacija senzora i aktuatora, počeo se pojavljivati skup novih sustava, čije ostvarenje donedavno nije bilo moguće. Riječ je o mrežama dinamičkih sustava, koje se javljaju u grupama mobilnih robota koji surađuju na ostvarivanju zajedničkog cilja; kolonama autonomnih vozila na automatiziranim autocestama; električnim mrežama nove generacije; adaptivnoj optici; pametnim konstrukcijama, na koje je ugrađen velik broj senzora i aktuatora, sa ciljem prigušenja neželjenih vibracija, ili upravljanja protokom fluida. Zajednička je karakteristika takvih sustava da se sastoje od relativno velikog broja prostorno distribuiranih dinamičkih podsustava, koji međusobno interagiraju fizičkim vezama i/ili komunikacijskim kanalima. Takvi se sustavi nazivaju mrežama dinamičkih sustava, ili kraće, dinamičkim mrežama.

Glavni je fokus današnjeg istraživanja u ovom području razvoj metoda sinteze upravljačkih zakona. Izazov je projektiranje lokalnih upravljačkih zakona koji će garantirati ostvarivanje globalnih ciljeva na razini cijele mreže. Trenutno, međutim, još uvijek ne postoji univerzalno primjenjiva teorija koja nudi fleksibino i robusno riješenje.

Osim teorije, trenutno nedostaje i programski okvir koji bi omogućavao razvoj, simulaciju, testiranje, i praktičnu implementaciju upravljačkih algoritama za takve sustave. U sklopu ovog rada razvijen je i predstavljen jedan takav programski okvir, koji omogućuje modeliranje velikog broja različitih vrsta dinamičkih mreža u Python programskom jeziku, a koji se izvršava unutar Robot Operating Systema (ROS).

Contents

List of Figures

List of Tables

1 Introduction

With the recent advances in communication and information technologies, and the development of new generations of sensors and actuators, a whole new set of systems started to emerge, which were unfeasible only years ago [1]. Some examples of such systems are: the so-called smart structures, composed of large amounts of sensors and actuators mounted on structural elements with the purpose of vibration dumping [2] or fluid flow control [3]; adaptive optics [4]; smart electrical power grids [5]; platoons of vehicles on automated motorways [6]; or large groups of ground mobile robots, unmanned aerial vehicles, or autonomous underwater vehicles, that collaborate towards a common goal [7],[8],[9]. The common characteristic of those systems is that they are composed of a relatively large number of spatially distributed dynamical subsystems, which interact with each other trough physical interconnections and/or communication links. Such systems are called networks of dynamical systems, or, shorter, dynamical networks.

The main focus of today's research in this area is the development of methods for control algorithm synthesis [1]. In most cases the challenge is not the design of subsystems, which can be very reliable when operating individually, but instead the design of local control laws, possibly in combination with distributed coordination schemes, for achieving common objectives at the network level. At the moment there is still no well understood, mature and widely applicable theory that offers scalable, robust, and reliable solutions to real-life network control problems. The research is currently still scattered, as problems of this nature are being explored also in biology, economics, sociology, game industry, etc. [10].

This thesis contributes to the field by introducing a software framework that can be used for development, simulation, and real-life implementation of control algorithm for networks of dynamical systems. The framework allows modelling networks in the Python language, while the framework itself runs within the Robot Operating System (ROS).

To motivate the performed work and to better understand of the problems, the thesis begins with a categorisation of dynamical networks, which shows that there is a wide variety of different cases. An overview of recent work and results in control of dynamical networks is also presented, to give the reader a starting point for further reading. The remaining sections present the architecture of the developed framework, its usage, and give several complete examples of networks implemented inside it.

The framework is available as free software, with the address for downloading it and licencing details given in section 4.

2 Distributed and decentralised control

This section will present a detailed classification of networks of dynamical systems by different criteria, and give an overview of some significant achievements in this area so far.

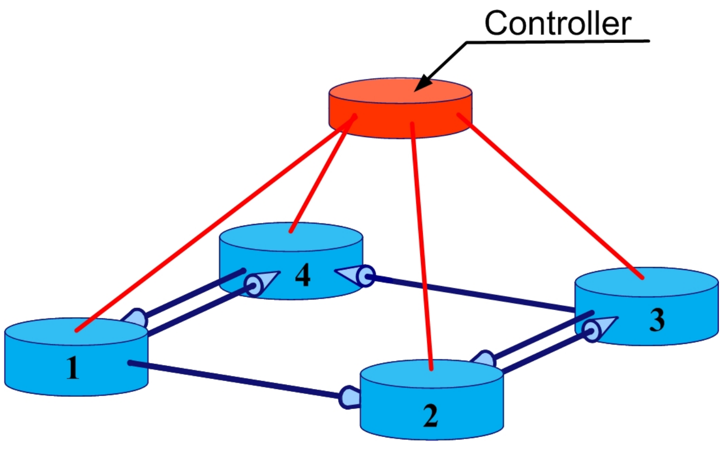
2.1 Classification of dynamical networks

To have a better understanding of the complexity of the field, and therefore to recognise the potential of the framework presented in this thesis, it is useful to be familiar with the different types of networks present in control systems today.

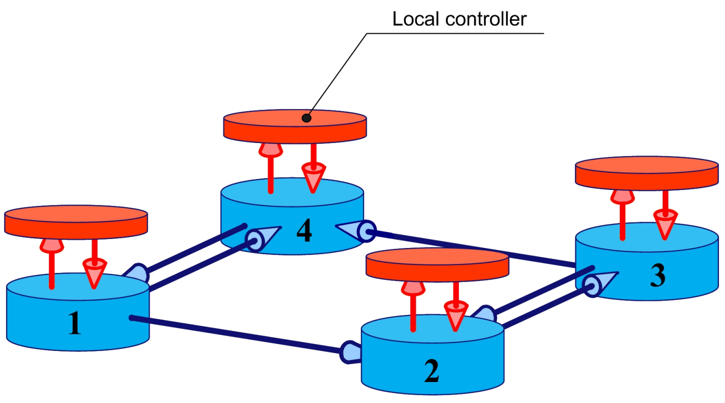
2.1.1 Control structures

In the past decade, it has been acknowledged that fully centralised control structures are not capable to cope with the complexity of large spatially distributed systems. A centralised control structure is one in which one central control unit collects all the measurements and sends commands to all actuators, as shown in Figure 2.1a. Such structures can guarantee globally optimal results of the controlled network, and their design is a mature field with a very well developed underlying theory. It has been proved that many practical problems with this structure can be formulated in terms of convex optimisation problems, and therefore efficiently solved [11]. However, some of their main limitations are that they are not scalable, not robust to failures, and are often practically impossible to implement.

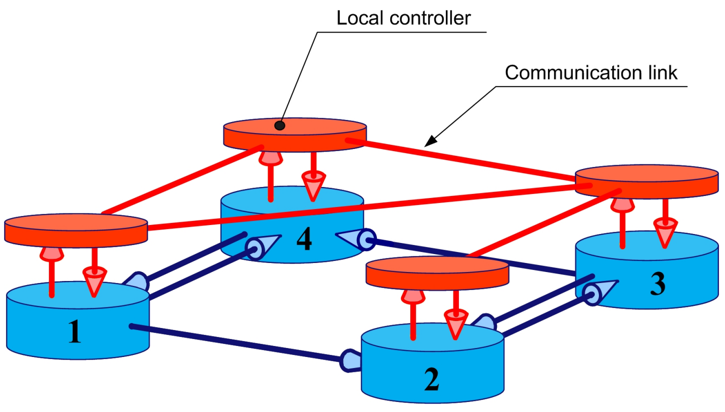
The exact opposite of centralised structures are decentralised ones, which are shown in Figure 2.1b. In this case every subsystem is controlled by a local controller, and there is absolutely no collaboration between local controllers. Their main advantage is that no long distance communication is required, which makes them theoretically



(a) Centralised control



(b) Decentralised control



(c) Distribbuted control

Figure 2.1: Control structures

ideal for practical implementation. They also have some significant disadvantages, like the possibility to offer only suboptimal solutions which could possibly be far from the global optimum, resulting in very inefficient behaviour. Using this structure the control synthesis is a non-convex problem, and there are no constructive algorithms for solving it [11].

The structure which has been widely recognised as most suitable for control of such large systems is the distributed structure, which is represented in Figure 2.1c. In such structure, each subsystem is controlled by a local controller which, apart from operating with locally available measurements, cooperates with a usually small set of neighbouring controllers. The controller communication network topology is in most cases required to be the same as the plant interaction network topology, but there could be exceptions.

2.1.2 Subsystem connection typology

The subsystems of a dynamical network can be physically coupled or decoupled. The former means that they directly affect each other's dynamics. Such systems range from inverted pendulums interconnected with springs, to complex electrical power grids, or the Internet.

If the systems are not physically coupled, it means that their interaction is through information exchange or measurements. They are connected by common goals or constraints, rather than physical links. Some examples are multirobot systems, vehicles platooning on motorways, and sensor networks.

2.1.3 Subsystem connection topology

Each subsystem has a certain set of neighbours, with which it interacts. This set can be either constant or change over time. In the former case the network topology is said to be static, while in the latter dynamic.

An example of subsystems that interact in a dynamic network topology are mobile robots which have a limited interaction range, due to limitations of sensing or communication equipment. This means that the set of each robot's neighbours is a function of their relative positions, which change over time.

2.1.4 Subsystem typology

The subsystems connected in a network can be all of the same type, or of different kinds. If the network is composed of identical systems, it is called homogeneous. Examples of such systems can be sensor networks or swarms of robots, given that all agents are of the same type. An interesting subset of such networks are spatially invariant networked systems, in which not only the subsystems are all equal, but also the dynamics of the system does not change moving along any spatial axis. An example are large segmented telescopes [12].

Networks composed of more than one type of subsystems are said to be heterogeneous. This is the case in the majority of applications, like electrical power grids, communication networks, or teams of different autonomous vehicles.

2.1.5 Control laws

The last important classification that is important to describe in this chapter is the difference of how the control laws of subsystems are being computed. The first possibility is that the controllers are predefined, which means that the control laws are computed offline, before operation. The second possibility is online solving of optimisation problems. In this case, each controller is computing its control output by solving an optimisation problem for every sampling period. This requires the controllers to iteratively exchange information between them inside one sampling period.

2.2 Overview of control synthesis approaches

Already in 1968, with the famous Witsenhausen (counter)example [6], it became clear that straightforward extension of the classical "centralized" control theory to distributed or decentralized control is not possible and that distributed control has some truly unique fundamental features, inherent limitations and are characterized with complex, sometimes counterintuitive, phenomena. The fact that distributed control problems are difficult and very challenging has later been supported with many results, like in [13] where some classes of structured control problems have shown to be intractable.

Recently, several structured control problems with some specific characteristics have been successfully studied, such as distributed control of linear spatially invariant systems [14], control of homogeneous systems interconnected over lattices or arbitrary symmetry groups (e.g. [15],[16]), and control of heterogeneous system interconnected over an arbitrary graph [17]. Other related and recent results include e.g. [11],[18],[19],[20]. In particular, in [11] the authors delineate the largest known class of structured control problems which can be formulated as convex optimisation problems, while in [19] the authors introduce the notion of spatially decaying operators in the insightful study of structural properties of optimal control problems with relation to the spatial structure of the problem.

Regarding stability analysis or synthesis of stabilising controllers for interconnected systems, a traditional and often used approach lies within the framework of dissipative dynamical systems which accounts for finding appropriately defined local storage functions, corresponding supply functions and the coupling conditions which together imply stability of the overall network. This approach has also been the underlying theoretical framework in recent results, e.g. [17],[21],[22]. Alternative approaches include the usage of vector or matrix Lyapunov functions [23]; approaches based on Youla parameterization of stabilizing controllers [24],[11]; or alternative approaches based on Nyquist-like “loop gain” conditions, as presented in [25],[26]. Over the past several years, the on-line optimization based control (Model Predictive Control (MPC), also referred to as receding horizon control) has shown its large potential for distributed control. Distributed MPC strategies have been presented in [27],[28],[29],[30].

3 Problem definition

Control of dynamical networks is a rapidly developing area of research, where most of the effort is put towards the development of a unified underlying theory. However, a universal software framework for development, simulation, testing, and practical implementation of such control algorithms is still missing. For this reason the transition from a general simulation to implementation on real systems requires big efforts and time investments. For example, it can be the case that people working on the theoretical development lack the programming skills required to implement complex distributed systems on specific hardware. Furthermore, even when having excellent programming skills and knowledge of embedded systems, this transition often requires a significant amount of time, having to reformulate the control algorithms in a form suitable for the particular system, and having to deal with communication protocols and limitations.

This has been the main motivation behind the work presented in this thesis. The goal was to explore the possibilities of creating a software framework that will allow development of distributed and decentralised control laws for both simulated and real networks of dynamical systems, minimising or completely removing the transition from one to the other. The framework should be fully flexible to accommodate all the possible characteristics of the dynamical network, which have been described in section 2.

A similar goal has already been achieved in robotics, by the Robot Operating System (ROS). The problem there was that a lot of robot control algorithms were being developed, but most of the time they were impossible to reuse across various systems and to connect with other existing software. Therefore, a lot of effort had to be put in the implementation process, often ending up rewriting existing software due to lack of portability. This is where ROS have contributed by offering a thin middleware which acts as a universal platform for software development, without being invasive (the software does not have to be designed specifically to work with ROS).

Since ROS already offers a standard systems to make different software communicate, and since there already exists a large amount of software packages for it, from hardware drivers to high level functionality, it has been decided to use it as the base for developing the dynamical network control framework.

4 The Dinsdale package

This section offers an overview of the Dinsdale package for the Robot Operating System (ROS). The goal of Dinsdale is to offer a convenient and unified framework to solve the problems which have been presented in section 3. It gives the possibility to implement control algorithms which can equally work on both simulated and real networks of systems, without the need to modify the code. Its flexibility also allows both the controllers and the plants to be interconnected in any desired way, allowing implementation of all control structures from section 2.

An additional characteristic is that networks of dynamical systems (controllers and simulated plants) are fully contained inside Python packages, completely independent from both Dinsdale and ROS. This allows separate development and testing, and encourages reuse of system models for any other purpose. The core of the Dinsdale framework itself is a ROS wrapper around those Python packages, which takes care of the execution, coordination, communication, and data logging, and which gives a standard interface for integration with other ROS software.

Python's popularity within the scientific community is increasing very rapidly, often at the expense of proprietary scientific computing software, making it the obvious choice for the end user side (system modelling). Python is, however, not only being used for implementation of dynamical networks, but the entire framework has been written in it. Its beauty and flexibility enabled the development of a powerful core functionality, while simultaneously keeping it relatively simple for users to understand, modify or extend.

ROS has been chosen as the underlying framework because it appears to be on its way to become the *de facto* standard in the robotics community for implementing control structures. This offers the possibility to integrate control algorithm developed in Dinsdale with other ROS packages, e.g. the ones for low-level control, localisation and mapping, or various virtual reality visualisation software. It is also possible to use standard ROS tools (e.g. `rostopic`, `rosbag`, or `rqt`) to inspect the dynamical network, or to store and reproduce data. Because it is important to have a basic understanding of ROS for further reading, a brief introduction and an explanation of its most important principles is given in Appendix A, along with references for further learning.

The Dinsdale package is free software, with all of its components released under the GNU General Public Licence (GPL) version 3, as published by the Free Software Foundation[1]. Dinsdale is hosted on GitHub, at `https://github.com/mross-22/dinsdale`.

4.1 Package structure

The Dinsdale package is a directory containing all the source code and data files required to run dynamical networks inside a ROS computational graph. The minimal structure of Dinsdale is illustrated in Figure 4.1. All the source files can be divided in three categories:

- System description $\longrightarrow$ `./src`
- Logged data $\longrightarrow$ `./bags`
- ROS wrapper $\longrightarrow$ everything else

The only files the user has to work with are in the system description category (inside `./src`). The Python package containing the entire definition of the dynamical network that is being simulated is `./src/dinsdale_system`. Inside it, the user defines the dynamics, control laws, and topology of the network. `./src` can contain more than one system, but only the one in the `./src/dinsdale_system` package is

[1] `http://www.gnu.org/licenses/gpl.html`

executed. This allows to store multiple systems, and choose the one to use by simply renaming its directory.

```
dinsdale
 |__ bags
 |__ msg
 |   |__ Floats.msg
 |__ scripts
 |   |__ node_controller
 |   |__ node_plant
 |   |__ node_simulation_time
 |   |__ node_topology_plants
 |   |__ result_analysis
 |   |__ setup_simulation
 |__ src
 |   |__ dinsdale_system
 |       |__ input_parameters
 |       |   |__ A_controllers.txt
 |       |   |__ A_plants.txt
 |       |   |__ controllers_iterative.txt
 |       |   |__ plants_topology.txt
 |       |   |__ system_types.txt
 |       |   |__ T.txt
 |       |   |__ x0.txt
 |       |__ tools
 |       |   |__ __init__.py
 |       |   |__ read_matrix.py
 |       |   |__ result_analysis.py
 |       |__ __init__.py
 |       |__ controller.py
 |       |__ plant.py
 |       |__ plants_topology.py
 |__ CMakeLists.txt
 |__ LICENSE
 |__ package.xml
 |__ README
 |__ setup.py
```

Figure 4.1: Dinsdale directory tree

Each time a simulation is set up, a directory containing the current date and time in its name is created inside `./bags`. In that subdirectory all the data logged during the simulation will be stored, inside ROS `.bag` files. The user does not necessarily need to access or manipulate those files, as Dinsdale has a script which allows data analysis directly in Python.

4.2 System definition

It has been mentioned in subsection 4.1 that the dynamical network is defined in a Python package called `dinsdale_system`, located inside the `src` directory of the main Dinsdale package (see Figure 4.1). This subsection will explain in detail how a network is defined. Several examples are presented in section 5. It is necessary to have some basic knowledge of graph theory terminology, which is given in Appendix B.

Note that all files and directories in this subsection will be referenced relatively to `./src/dinsdale_system` for the sake of avoiding long path names.

4.2.1 Plants

Each plant in the network is defined by a `Plant` class. This class is located inside `plant.py`, and its structure is shown in Figure 4.2. It is necessary to define as many classes, each in its own file, as the number of different types of plants in the network.

Example 4.1: A group of ten identical robots will need a single `plant.py` file.

Example 4.2: A group composed of two identical autonomous underwater vehicles (AUVs), one autonomous surface vehicle (ASV), and three identical unmanned aerial vehicles (UAVs) need to have three plant implementations: `plant.py`, `plant_1.py`, and `plant_2.py`.

Table 4.1 explains what the attributes of the class are. All, except for `n` and `T`, are of the `numpy.matrix` type. They need always to have the shape of column vectors, but their length is not limited.

As shown in Figure 4.2, the `Plant` class has three methods:

- `__init__(n, x0, T)` – The initialisation of a `Plant` instance. It initialises all the attributes, and executes the initialisation code set by the user. This method is called only once, at the beginning of the execution.

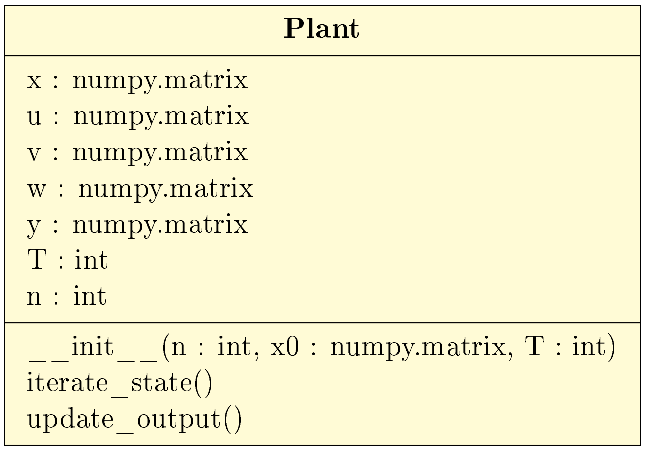


Figure 4.2: Plant class

Table 4.1: Plant attributes

Plant	
x	plant states
u	input from controller
y	output for controller
w	input from other plants
v	output for other plants
n	ordinal number of the node
T	sample time

- `iterate_state()` – The user here sets the equations for the update of the plant's states `x` and output for other plants `v`. This method is executed when new inputs from the controller are received.
- `update_output()` – This method is executed when the new set of input from neighbouring plants `w` is received, and its purpose is to contain the equation for updating the output for the controller `y`.

4.2.2 Controllers

Each controller in the network is defined by a `Controller` class. Following the same logic applied to plants, the class for a controller is inside a `controller.py` file. In case of multiple different types of controllers, there will be multiple files, called `controller.py`, `controller_1.py`, `controller_2.py`, etc.

Table 4.2 explains what the attributes of the `Controller` class are. Except for `n`, `T`, and `finished`, the rest are of the `numpy.matrix` type. They need always to have the shape of column vectors, but their length is not limited.

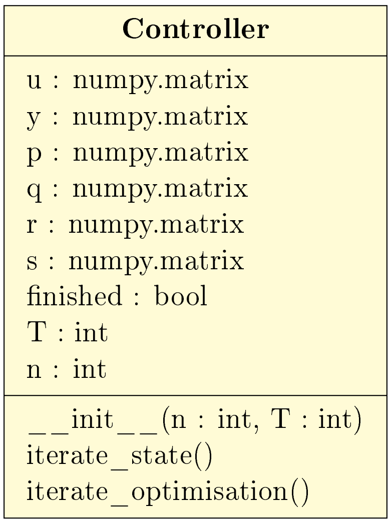


Figure 4.3: Controller class

Table 4.2: Controller attributes

Controller	
y	input from plant
u	output for plant
q	input from other controllers
p	output for other controllers
s	iterative input from other controllers
r	iterative output for other controllers
finished	iterative communication finished
n	ordinal number of the node
T	sample time

As shown in Figure 4.3, the `Controller` class has three methods:

- `__init__(n, T)` – The initialisation of a `Controller` instance. It initialises all the attributes, and executes the initialisation code set by the user. This method is called only once, at the beginning of the execution.
- `iterate_state()` – The user here sets the equations for the update of the controller's outputs, both for the plant and for neighbouring controllers (`u` and `p`). This method is executed when new data from the plant (`y`) is received.
- `iterate_optimisation()` – This method is executed after `iterate_state()` if the controllers communicate iteratively within one simulation step. This method is usually used when the control law is computed by collaboratively solving an optimisation problem on-line. If this is the case, the vectors used for iterative communication are `r` and `s`. In each simulation step this method is called until `finished` is set to `True`. In the next iteration it will automatically be reset to `False`.

4.2.3 Plants interaction topology

The topology of the plants interaction network can be static or dynamic. When it is static, nothing has to be done. In case it is dynamic, the user has to determine the law according to which the topology is changing. This is done in the `DynamicPlantsTopology` class (shown in Figure 4.4), inside the `plants_topology.py` file (note that there can be only one such file for a network).

Example 4.3: If the plants are mobile robots with limited interaction range, then the topology is a function of their relative positions.

The class `PlantsTopology` initialises:

- `A` – The adjacency matrix of the plants interaction network (a square $(n \times n)$ matrix, where n is the number of plants).
- `nodes` – The number of plants.
- `w` – The output from each plant (a list of n elements, each being a one dimensional `numpy.array`).

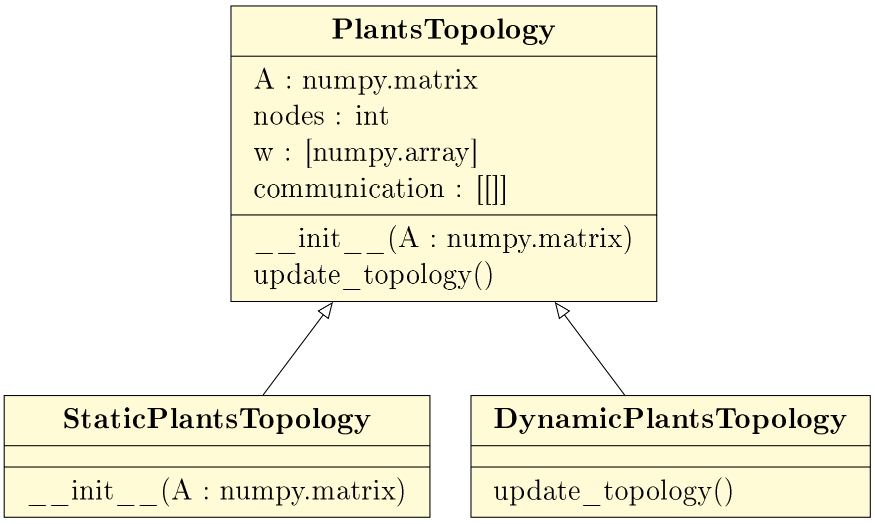


Figure 4.4: Plants topology classes

- `communication` – A list of neighbours of each plant (a list of n lists). At this point its empty.

The class `StaticPlantsTopology` inherits from `PlantsTopology` and extends the initialisation by populating the `communication` list. This is done by looping through the adjacency matrix. Once the neighbours of each plant are found, in each simulation step the plant will receive their outputs.

The class `DynamicPlantsTopology` also inherits from `PlantsTopology`, but it implements the method `update_topology()`. This method is empty by default, and this is where the user can define the laws according to which the topology is changing, in case it is dynamic. The laws could use the data sent by plants, or be functions of time. Another possible application of the `update_topology()` method, apart from having a dynamic topology, is multiplying the values exchanged by plants with some weights, which can also change over time. Whatever its usage may be, the objective in this method is to fill the list `communication` with lists of neighbours of each plant, in every time step.

4.2.4 Parameters

Inside `./input_parameters` there are seven items to be set:

- `A_controllers.txt` – The adjacency matrix of the controller communication network. The file can be empty if the controllers do not communicate.
- `A_plants.txt` – The adjacency matrix of the plant communication network. Even if the plants do not interact, it has to be of size $(n \times n)$, where n is the number of plants, but filled with zeros.
- `controllers_iterative.txt` – Determines whether the controllers communicate iteratively inside one time step (1 if they do, 0 or empty otherwise).

- `plants_topology.txt` – 0 if the plants topology is static, 1 if dynamic.
- `system_types.txt` – Describes the type of controller and plant for each subsystem. It is a matrix of size $(n \times 2)$, where n is the number of subsystems. The first column corresponds to controllers, the second to plants.
- `T.txt` – A (3×1) vector, the elements of which correspond to the sampling period (used for discretisation of the system dynamics), the real duration of a simulation time step, and the final time of the simulation, all given in seconds.
- `x0.txt` – Contains the initial conditions for each plant. It is an $(n \times m)$ matrix, where n is the number of plants and m the number of states of a plant (in case of heterogeneous systems, the plant with the largest dimension of the state space determines this number).

4.2.5 Tools

The additional package `./tools` contains useful tools which are not part of the dynamical network. Two modules can be find inside it:

- `read_matrix.py` – This is the function all other modules use for reading matrices from files. It is placed here to make it available also for the user, in case he or she wants to load some additional matrices.
- `result_analysis.py` – This module is where all data from a simulation is made available to the user for analysis of any kind.

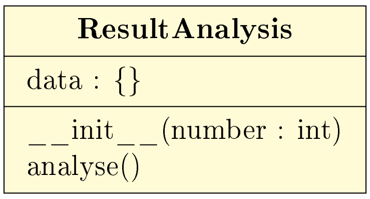


Figure 4.5: Result analysis class

The `result_analysis.py` file contains the `ResultAnalysis` class, shown in Figure 4.5. In its initialisation method, the class initialises a dictionary called `data`. This dictionary gets filled by a core Dinsdale class with data stored during a simulation. The keys of the dictionary correspond to names of controller and plant attributes. The value of a key is a list of a list of all values of the specific attribute of every node in the network. In the method `analyse()` the user is free to manipulate this data in any way, e.g. plotting it.

4.3 Runtime

The `dinsdale_system` package is a pure Python package, which contains the description of a dynamic network. To generate as many instances of its classes as required by the network specification, initialise their values, transform them in ROS nodes, set up the communication networks, and coordinate the execution, there is a wrapper around it. This wrapper is the core Dinsdale functionality.

Figure 4.6 shows in a simplified manner the classes inside the Dinsdale package. It can be seen how each class in `dinsdale_system` is being use by another one, outside of `dinsdale_system`.

At runtime, Dinsdale sets up as many ROS nodes as defined by the user, and connects them accordingly in a ROS computational graph. Figure 4.7 shows how the graph looks like for a single subsystem of a network, composed of a plant and its controller. The oval shapes represent ROS nodes, the smaller rectangles are ROS topics, and the big rectangles are called namespaces. Inside each node there is an instance of the user defined `Plant` or `Controller` classes.

Figure 4.8 illustrates a minimal graph for the simulation of a network of two subsystems. In this example the control laws are decentralised, which means the controllers do not communicate. It is important to observe the central node called

`/plants_topology`, through which all interaction between plant are passing. As it has been shown in subsubsection 4.2.3, the topology of plant interactions can be dynamic, and the user is free to define its rules. For this reason it is more convenient to control the topology from a central place, rather than having to modify each plant node in every iteration. This is possible because the interactions, like the plants themselves, are simulated. If simulated plants were to be replaced by physical ones, interactions would also become physical, and would not need to be simulated.

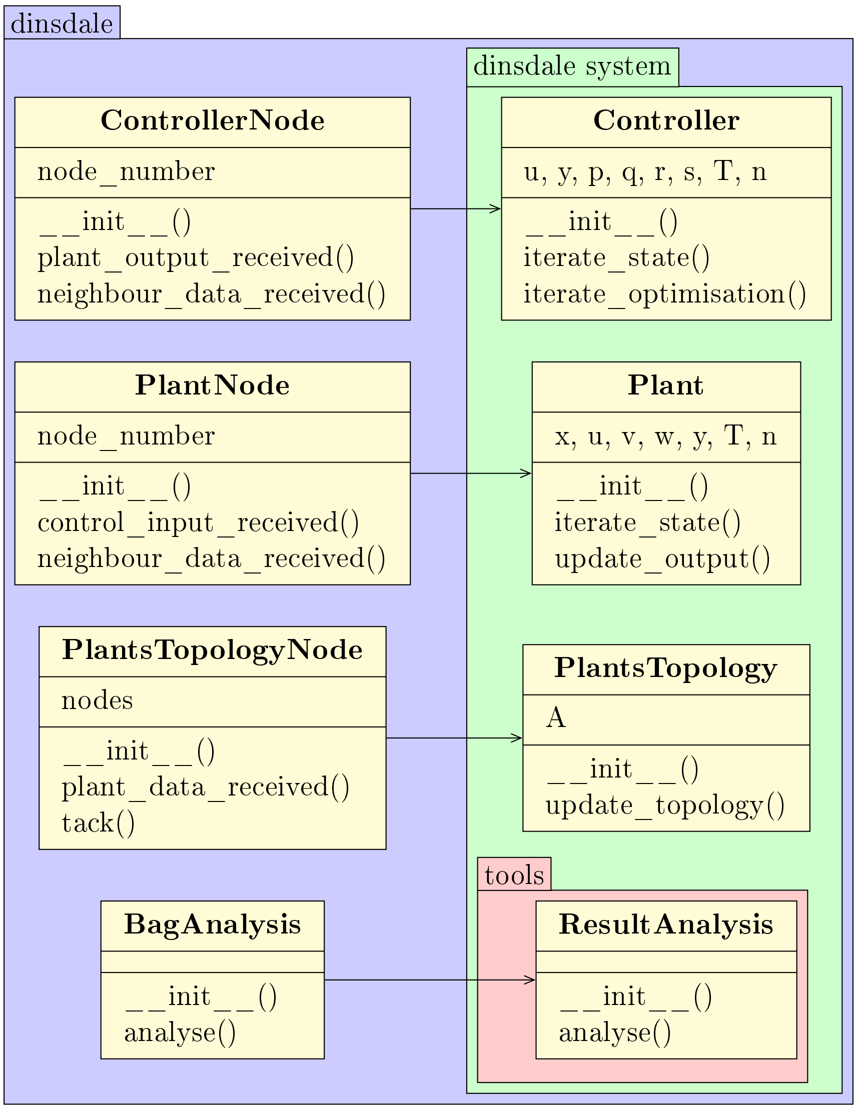


Figure 4.6: Simplified Dinsdale class diagram

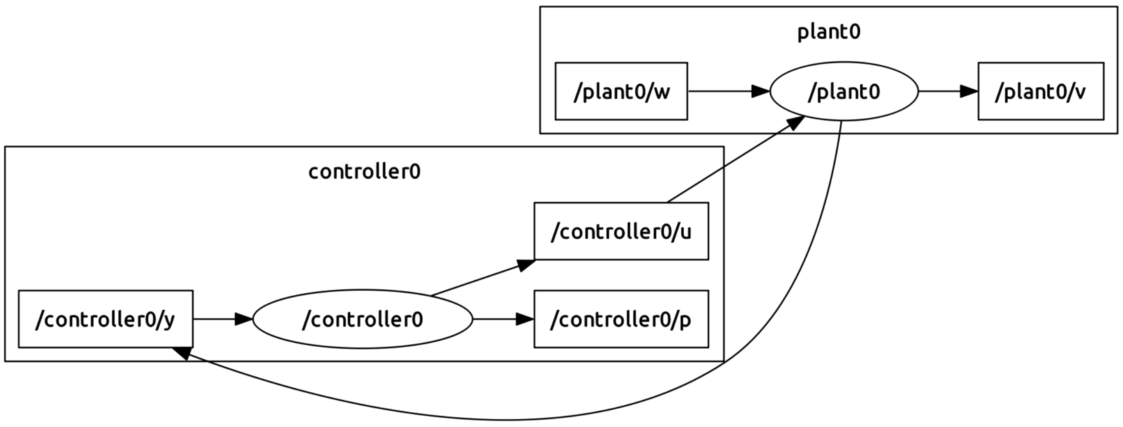


Figure 4.7: ROS graph for a single subsystem

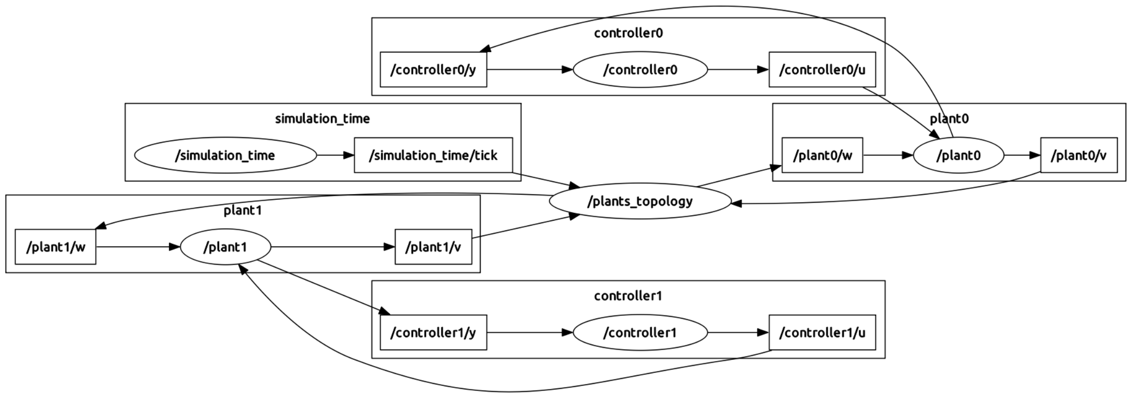


Figure 4.8: ROS graph for a network with 2 subsystems

In the simulation there is also a `simulation_time` namespace. It contains a node that publishes ticks to a topic, which gives the time to the simulation, with the user defined speed.

An example of a simulation of a network with three subsystems can be seen in Figure 4.9. In this case the controllers are communicating through `~/p` topics, and their communication does not pass through a central node. The drawback of not having a central node is that the topology of the network of controllers has to be static.

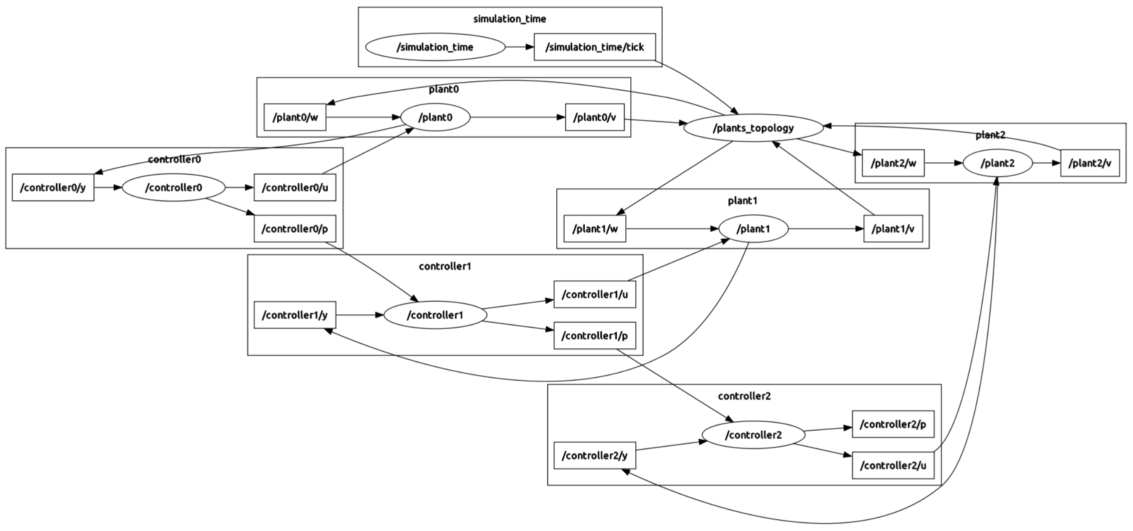


Figure 4.9: ROS graph for a network with 3 subsystems

After all nodes are set up in a ROS graph and the simulation time is started, Dinsdale starts simulating the network using the classes defined in `dinsdale_system`. Figure 4.10 shows the order in which the methods are being called inside one time step, when simulating a network of two systems. The dashed arrows represent optional communication between controllers (single and iterative). When the controllers are not communicating iteratively, the method `Controller.iterate_optimisation()` is not called at all. The letters near the arrows represent the name of the attribute that is being sent.

4.4 Usage

This subsection illustrates how to use Dinsdale once the `dinsdale_system` package containing the dynamical network is set up accordingly to subsection 4.1.

If Dinsdale is not already installed, the simplest way to do so is by placing the package in the local `catkin` workspace and inside it run:

```
$ catkin_make
```

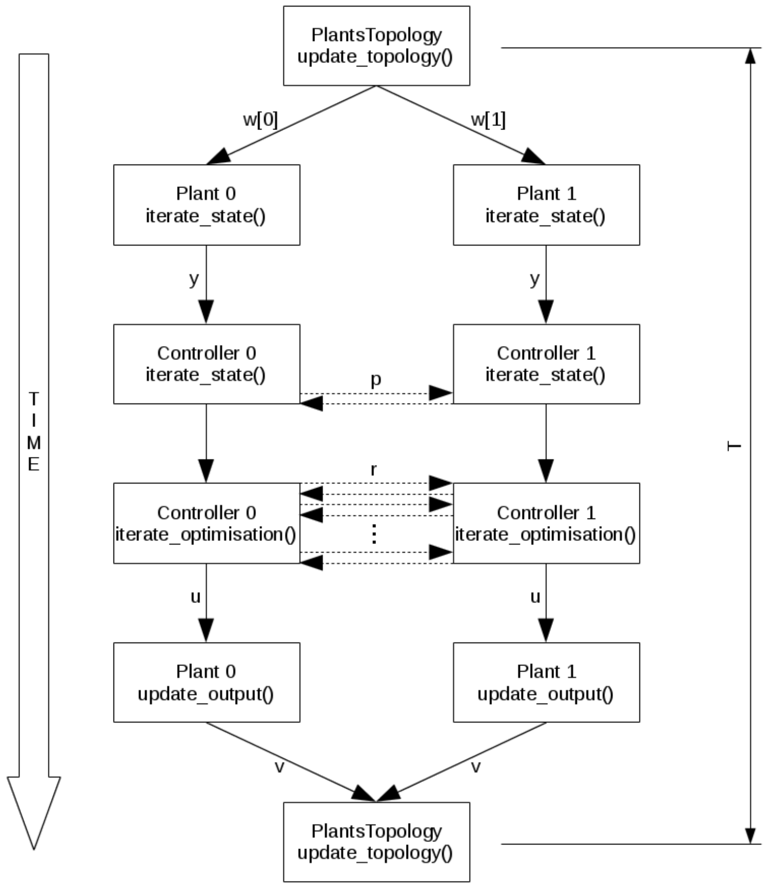


Figure 4.10: Execution flow

For more details see the references given in Appendix A.

4.4.1 Executing the simulation

Before starting the simulation, it is necessary to run the ROS Master:

```
$ roscore
```

Once the Master is up, the simulation can be set up executing (in a new terminal window):

```
$ rosrun dinsdale setup_simulation
```

`setup_simulation` is a script located in `./scripts`, which reads all the data from `./src/dinsdale_system/input_parameters`, generates a ROS `.launch` file accordingly, and runs it. The generated file is called `simulation.launch`, and is located in the root directory of Dinsale. It can be used in future to run the same simulation, which can be useful if Dinsdale is used as a part of a bigger software system.

When executed (either automatically or manually), `simulation.launch` will create all the necessary ROS nodes for the given system, and connect them in a ROS graph. The graph can be visualised using the `rqt_graph` tool, which gives an output similar to those shown in Figure 4.7, Figure 4.8, and Figure 4.9. At this point the terminal window is waiting for the user to press `Enter` to run the simulation. When `Enter` is pressed, the `/simulation_time` node starts publishing ticks to the `/simulation_time/tick` topic, which acts as the simulation clock, determining the frequency of the iterations described in Figure 4.10.

When the simulation ends, the `/simulation_time` node stops, while the rest remain active and have to be interrupted manually (`Ctrl-C`).

While the simulation is running, it is possible to analyse the network (e.g. monitor the topics in real time) using all the standard ROS tools, like `rostopic` or `rqt_plot`.

Each ROS node can be also ran individually, using `rosrun` on one of the scripts located inside `./scripts`. Some of them require passing certain arguments, but this can be seen at the beggining of each script.

4.4.2 Result analysis

When a simulation is started, it automatically creates a directory inside `./bags`, called `<date>_<time>_dinsdale`. Inside it each controller and plant will log the values published on topics, in the format of ROS `.bag` files. Dinsdale provides a tool for analysis of their content in Python. The user defines the analysis to be done in the `./src/dinsdale_system/tools/result_analysis.py` file, as described in subsubsection 4.2.5. To execute it, automatically loading the `.bag` file from the last simulation, the user has to run the `./scripts/result_analysis` script as:

```
$ rosrun dinsdale result_analysis
```

Optionally, if the analysis has to be performed on data from another simulation, the user has to pass to the script the name of the directory containing the `.bag` files:

```
$ rosrun dinsdale result_analysis <directory>
```

Storing data in `.bag` files is very convenient not only for analysis, but also because the `rosbag` tool allows reproducing their content in the same order it was stored. This means that if some tests have been made using a real system, which could not be available at all times, it is possible to reproduce the logged data at any time, as if it the system was connected.

4.4.3 Replacing simulated plants with real systems

One of the main motivations for the development of the Dinsdale package was to make the transition from simulation to implementation as simple as possible. This was kept in mind during the entire design process, which can be seen for example in the structure of controller nodes, which have all their input and output topics set within their namespaces (visible in Figure 4.7. This makes them agnostic about who is publishing their inputs and who is using their outputs. Not only this gives the possibility to replace the simulated plants with real systems, but also to use Dinsdale together with the big variety of existing ROS packages, e.g. for simultaneous localisation and mapping, visualisation, image processing, etc.

As ROS is gaining popularity, more and more hardware drivers are available as ROS packages. At the same time, small single-board ARM computers are rapidly being adopted as cheap and efficient control units for a huge variety of systems. Since those computers are able to run GNU/Linux distributions, and therefore ROS, these facts makes it very convenient to use Dinsdale generated controllers for control of real networks of systems.

5 Examples

This section will present several examples of networked dynamical systems from different areas, each specifically chosen to demonstrate some of the various possibilities of the Dinsdale package. The examples are taken from works by various authors, which can be found in the references. Note that, even where the mathematical models are given in continuous time, all Dinsdale implementations use the corresponding discretized models.

The Python packages with the full code for each example can be found inside the `src` directory of the Dinsdale package. To run any of them inside Dinsdale, it is only necessary to rename the desired package to `dinsdale_system` and start the simulation.

Note that all files and directories in this section will be referenced relatively to `./src/dinsdale_system` for the sake of avoiding long path names.

5.1 Interconnected inverted pendulums

This example illustrates decentralised control of two physically coupled dynamical systems – inverted pendulums interconnected with a spring.

5.1.1 System description and control objectives

The system is composed of two identical interconnected inverted pendulums, as described in [32], and shown in Figure 5.1. The two pendulums are coupled by a spring that is able to slide up and down the rods, and the position of which is unknown. In this example the position of the spring can be a discontinuous function of time and the system state. This is an example commonly used as a benchmark for decentralised control algorithms. The control objective is to stabilise both pendulums in their vertical positions.

5.1.2 Mathematical problem formulation

The system can be represented with a time invariant linear part, perturbed by an additive nonlinearity which depends on both time and the state:

$$\dot{x}_i = A_i x_i + B_i u_i + B_{w_i} w_i(t, x), \qquad i \in N, \tag{1}$$

where N is the number of subsystems, and where the uncertain functions w_i are the interconnections between subsystems. The only information about those interconnection functions is that they must satisfy the quadratic constraint:

$$w_i^T w_i \leq \alpha_i^2 x^T W_i^T W_i x, \tag{2}$$

where $\alpha_i > 0$ is the bounding parameter, and W_i is a constant $l \times n$ matrix, with n being the dimension of x.

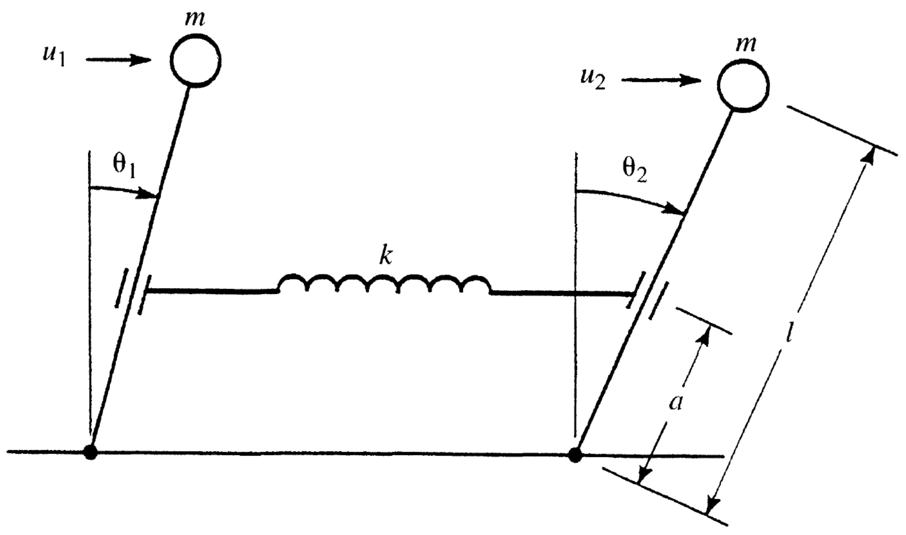


Figure 5.1: Two interconnected pendulums [31]

The dynamics of this particular system are described in [31] as:

$$\begin{aligned}
\dot{x}_1 &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} x_1 + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_1 + e \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix} x_1 + e \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} x_2 \\
\dot{x}_2 &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} x_2 + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_2 + e \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} x_1 + e \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix} x_2,
\end{aligned} \tag{3}$$

where $x_i = \begin{bmatrix} \theta_i & \dot{\theta}_i \end{bmatrix}^T$, $i \in \{1, 2\}$. It is assumed that the spring can randomly slide up and down the two rods, with jumps of an unpredictable size and direction. This dynamical interconnection is described with the parameter $e(t, x) : \mathbb{R}^5 \to [0, 1]$.

The objective is to compute the decentralised control laws:

$$\begin{aligned}
u_1(x_1) &= K_1 x_1 \\
u_2(x_2) &= K_2 x_2,
\end{aligned} \tag{4}$$

to robustly stabilise the system for any value of $e \in [0, 1]$.

5.1.3 Control synthesis

In [31] a control synthesis approach for robust stabilisation of nonlinear systems has been presented, within the framework of linear matrix inequalities (LMI). The system has to be described as an uncertain linear time invariant system (1).

The problem consists in finding the block diagonal gain matrix $K = \mathrm{diag}(K_1, K_2)$ and a function $V(x)$ which satisfies the following condition for any state $x \in \mathbb{R}^n$ of the *closed loop* system:

$$\alpha_1(\|x\|) \leq V(x) \leq \alpha_2(\|x\|), \tag{5}$$

where $\alpha_i \in K_\infty$ are Hahn's functions (for a detailed definition see [33]). The stability can be then defined by negative definiteness of the derivative:

$$\dot{V}(x) = (\nabla_x V(x))^T \left((A + BK)\, x + B_w w(t, x) \right) \leq -\alpha_3(\|x\|). \tag{6}$$

In [31] the authors use a quadratic Lyapunov function:

$$V(x) = x^T P x, \tag{7}$$

where P is a block diagonal matrix $P = \operatorname{diag}(P1, P2)$.

Under this assumption of a block diagonal Lyapunov matrix P, it is shown in [31] that (5) and (6), when combined with the S-procedure [34], can be transformed into LMIs, and therefore efficiently solved.

For this particular example, the resulting solution are the following gain matrices:

$$\begin{aligned} K_1 &= \begin{bmatrix} -3.008 & -3.0032 \end{bmatrix} \\ K_2 &= \begin{bmatrix} -3.008 & -3.0032 \end{bmatrix}. \end{aligned} \tag{8}$$

5.1.4 Implementation

The first step to implement this example is to write the dynamical model of the plants. Since both plants are equal, it is enough to define one model, inside `plant.py`. It is first necessary to initialise the model, placing the corresponding matrices, shown in Listing 5.1, inside the `__init__()` method. The equation describing the plant's dynamics, from Listing 5.2, are then placed in the `iterate_state()` method. In this method it is important to set variables `self.x` and `self.v`, as they will be the future plant state and the influence to neighbour plants. Note that the interconnection parameters e are random numbers, which change in each iteration. Inside the `update_output()` method goes the equation for updating the output value sent to the controller, `self.y`, which is shown in Listing 5.3. Variable `self.w` is the influence from neighbouring plants.

Listing 5.1: Plant initialisation

```
self.A = np.matrix([[1, T],
                    [T, 1]])
self.B = np.matrix([[0, T]]).T
self.H_this = np.matrix([[0, 0],
                         [-T, 0]])
self.H_other = np.matrix([[0, 0],
                          [T, 0]])
```

Listing 5.2: Plant state update

```
e = np.random.rand()
self.x = self.A*self.x + self.B*self.u + self.H_this*self.x +
    self.H_other*self.w
self.v = self.x
```

Listing 5.3: Plant output update

```
self.y = self.x
```

The control law has to be defined inside `controller.py`. It is again the case that, because the controllers are identical, it is enough to define only one. The controller initialisation consists only of initialising the gains (Listing 5.4), which has to be done inside the `__init()__` method. The control law (Listing 5.5) is then written inside `iterate_state()`, to compute `self.u`.

Listing 5.4: Controller initialisation

```
self.K = np.matrix('-3.008 -3.0032')
```

Listing 5.5: Controller update

```
self.u = self.K*self.y
```

After the system is set up, the last thing to do is to define the parameters inside the `./input_parameters` directory.

- `A_controllers` – This should be empty, as the controllers do not communicate.
- `A_plants` – The adjacency matrix describing the plant interaction is $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.
- `controllers_iterate` – This should also be empty, as the controllers do not communicate at all.

- `system_types` – All the systems and the controllers are of the same type (type 0), which is defined as $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$.
- `T` – This simulation uses a sampling period of 0.01 s, and that can be also used for the real duration of a simulation step[2]. The final time is chosen to be 10 s. The vector is therefore $\begin{bmatrix} 0.01 \\ 0.01 \\ 10 \end{bmatrix}$.
- `x0` – The initial conditions chosen in this example are $\begin{bmatrix} -0.2 & 2 \\ 0.4 & 1 \end{bmatrix}$. This means that $x_1(0) = \begin{bmatrix} -0.2 & 2 \end{bmatrix}^T$ and $x_2(0) = \begin{bmatrix} 0.4 & 1 \end{bmatrix}^T$.

One additional thing, which is not necessary to run the simulation, and which can be done afterwards, is to write the code for analysis of the simulation results. To obtain the same plots as here, for example, the code from Listing 5.6 should be placed in `./tools/result_analysis.py`, inside the `analyse()` method.

Listing 5.6: Plotting results

```
t = np.arange(0, 10, .01)
p0 = np.array([x[0:2] for x in self.data['y'][0]])
p1 = np.array([x[0:2] for x in self.data['y'][1]])

plt.figure(1)
plt.plot(t, p0[:, 0], label = 'plant 0')
plt.plot(t, p1[:, 0], label = 'plant 1')
plt.legend()
plt.xlabel('t [s]')
plt.ylabel('[x_i]_1 [rad]')

plt.figure(2)
plt.plot(t, p0[:, 1], label = 'plant 0')
```

[2]These values do not have to be the same as here, especially the duration of a simulation step, which should be set to a much longer time if it is desired to monitor some topics in real time, using `rostopic` or `rqt_plot`.

```
plt.plot(t, p1[:, 1], label = 'plant 1')
plt.legend()
plt.xlabel('t [s]')
plt.ylabel('[x_i]_2 [rad/s]')

plt.show()
```

5.1.5 Simulation results

Figure 5.3 shows the response of the network. In Figure 5.3a it is possible to see the response of the first state variable of both pendulums, while in Figure 5.3b the response of the second. It is visible that the decentralised control law successfully stabilises the network.

The runtime graph of this dynamical network, generated by the Dinsdale package to simulate it, can be seen in Figure 5.2.

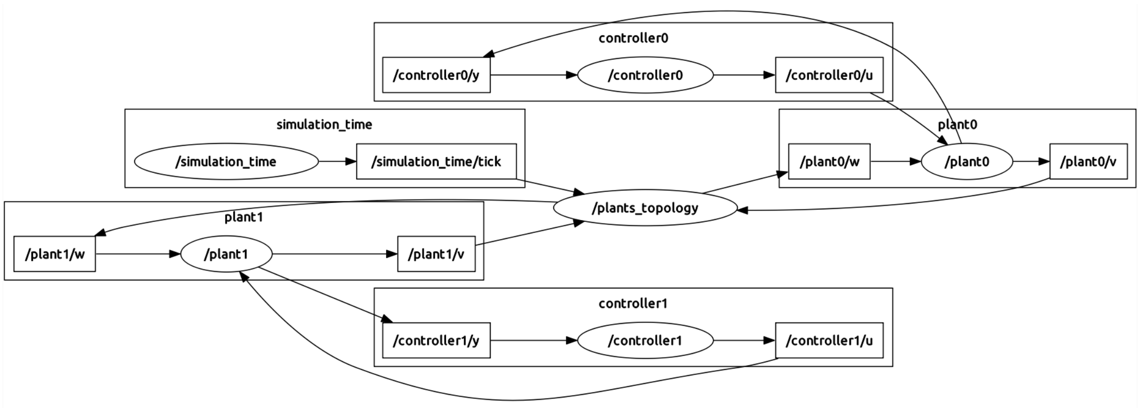
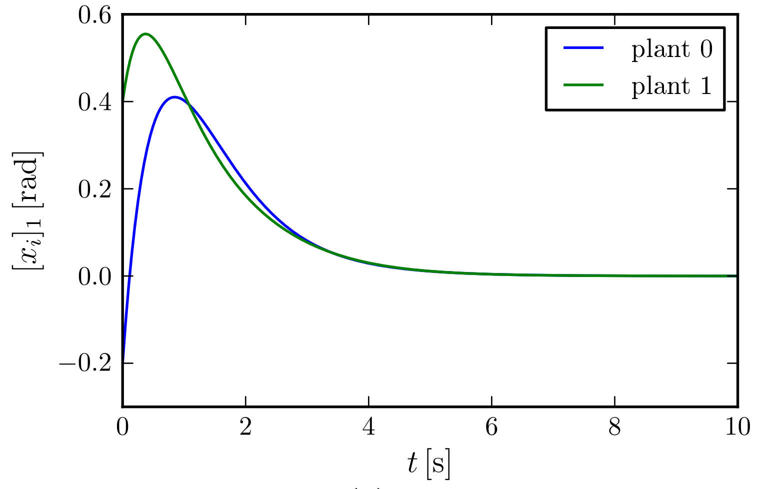
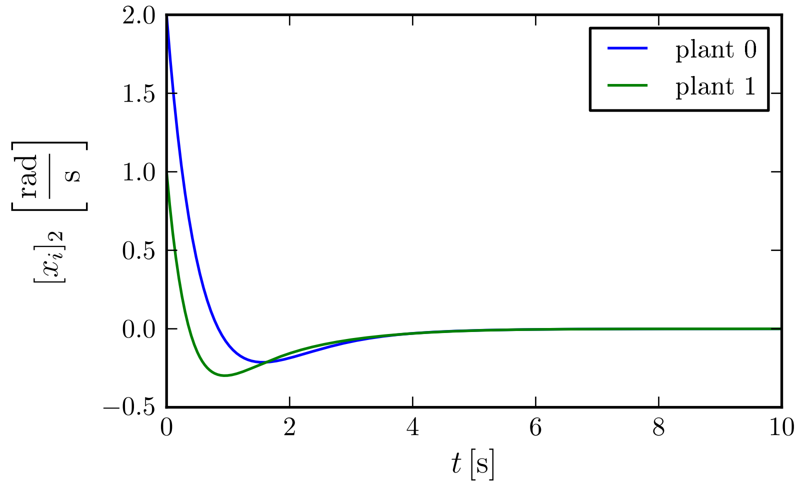


Figure 5.2: Runtime graph



(a) Angles



(b) Angular velocities

Figure 5.3: System response

5.2 A heterogeneous network of systems

This example illustrates decentralised control of a heterogeneous network of three physically coupled systems.

5.2.1 System description and control objectives

The dynamical network in this example is composed of three subsystems, each one different from the others. The control objective is to achieve stability through decentralised control laws. Like the example presented in subsection 5.1, this one also comes from [31], where more in-depth analysis can be found.

5.2.2 Mathematical problem formulation

The dynamics of the three subsystem are again represented as uncertain linear time invariant systems, accordingly to (1), with interconnections constrained by (2). For this example the equations are as follows:

$$\begin{aligned}
\dot{x}_1 &= \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} x_1 + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_1 + e_{13} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} x_3 \\
\dot{x}_2 &= \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix} x_2 + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_2 + e_{23} \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} x_3 \\
\dot{x}_3 &= \begin{bmatrix} 0 & 1 \\ 3 & 4 \end{bmatrix} x_3 + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_3 + e_{31} \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix} x_1 + e_{32} \begin{bmatrix} 1 & 5 \\ 4 & 6 \end{bmatrix} x_2.
\end{aligned} \tag{9}$$

The objective is to compute the decentralised control laws:

$$u_i(x_i) = K_i x_i, \qquad i \in \{1, 2, 3\}. \tag{10}$$

5.2.3 Control synthesis

The approach to the synthesis of control laws is the same as described in subsubsection 5.1.3, with the only difference that, due to the restrictive formulation of K being a block diagonal matrix, it is not possible to *a priori* set the bounds $\alpha_i = 1$ of the uncertain interconnection functions $e_{ij}(t, x)$ because the problem becomes unfeasible. The α_i are instead treated as variables.

Removing the constraints on bounds α_i, it is possible to compute the overall block diagonal gain matrix $K = \text{diag}(K_1, K_2, K_3)$, obtaining the following values:

$$\begin{aligned} K_1 &= \begin{bmatrix} -5.88 & -6.66 \end{bmatrix} \\ K_2 &= \begin{bmatrix} -0.67 & -1.48 \end{bmatrix} \\ K_3 &= \begin{bmatrix} -6.67 & -8.82 \end{bmatrix}, \end{aligned} \tag{11}$$

which guarantee robust stability of system (9) for the following bounds of the interconnection functions:

$$\begin{aligned} |e_{13}| &\leq 0.6481, & |e_{23}| &\leq 0.4007, \\ |e_{31}| &\leq 0.2192, & |e_{32}| &\leq 0.2192. \end{aligned} \tag{12}$$

5.2.4 Implementation

This example deals with a heterogeneous network of three subsystems, and therefore there have to be three implementations of the `Plant` class, and three of the `Controller` class. This means that there are going to be three files for the plants, and three for the controllers. Their implementations are described in Table 5.1.

Note that it would have been possible to design the network using only a single file for the controllers, and a file for the plants, since each node at runtime knows its ordinal number. In that case the files would contain information about the entire network, and the nodes would select their behaviour at runtime. This has not been done in order to show an example of implementation of heterogeneous systems, and to maintain the simplicity of the code.

Table 5.1: Heterogeneous network definition

(a) Plants

Plant	`__init__()`	`iterate_state()`	`update_output()`
`plant.py`	Listing 5.7	Listing 5.8	Listing 5.9
`plant_1.py`	Listing 5.10	Listing 5.11	Listing 5.9
`plant_2.py`	Listing 5.12	Listing 5.13	Listing 5.9

(b) Controllers

Controller	`__init__()`	`iterate_state()`
`controller.py`	Listing 5.14	Listing 5.15
`controller_1.py`	Listing 5.16	Listing 5.15
`controller_2.py`	Listing 5.17	Listing 5.15

Listing 5.7: Plant 0 initialisation

```
self.A = T*np.matrix('0 1; 2 3')
self.B = T*np.matrix('0; 1')
self.H_13 = T*np.matrix('1 2; 3 4')
```

Listing 5.8: Plant 0 state update

```
e_13 = .4
self.x = self.A*self.x + self.B*self.u + e_13*self.H_13*self.w + self.x
self.v = self.x
```

Listing 5.9: Plant 0, 1, and 2 output update

```
self.y = self.x
```

Listing 5.10: Plant 1 initialisation

```
self.A = T*np.matrix('0 1; -1 -2')
self.B = T*np.matrix('0; 1')
self.H_23 = T*np.matrix('1 0; 2 1')
```

Listing 5.11: Plant 1 state update

```
e_23 = .33
self.x = self.A*self.x + self.B*self.u + e_23*self.H_23*self.w + self.x
self.v = self.x
```

Listing 5.12: Plant 2 initialisation

```
self.A = T*np.matrix('0 1; 3 4')
self.B = T*np.matrix('0; 1')
self.H_31 = T*np.matrix('3 0; 2 1')
self.H_32 = T*np.matrix('1 5; 4 6')
```

Listing 5.13: Plant 2 state update

```
e_31 = .15
e_32 = .1
self.x = self.A*self.x + self.B*self.u + e_31*self.H_31*self.w[:2, :] +
    e_32*self.H_32*self.w[2:, :] + self.x
self.v = self.x
```

Listing 5.14: Controller 0 initialisation

```
self.K = np.matrix('-5.88 -6.66')
```

Listing 5.15: Controller 0, 1, and 2 update

```
self.u = self.K*self.y
```

Listing 5.16: Controller 1 initialisation

```
self.K = np.matrix('-0.67 -1.48')
```

Listing 5.17: Controller 2 initialisation

```
self.K = np.matrix('-6.67 -8.82')
```

The parameters inside the `./input_parameters` directory for this simulation are:

- `A_controllers` – This should be empty, as the controllers do not communicate.
- `A_plants` – The adjacency matrix of the plant interaction is $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$.

- `controllers_iterate` – This should also be empty, as the controllers do not communicate.
- `system_types` – There are three types of plants, with a type of controller for each. This is described as $\begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$.
- `T` – This simulation uses a sampling period of 0.01 s, and that value is also used for the real duration of a simulation step. The final time is chosen to be 10 s. The vector therefore is $\begin{bmatrix} 0.01 \\ 0.01 \\ 10 \end{bmatrix}$.
- `x0` – The initial conditions chosen in this example are $\begin{bmatrix} 0.5 & 2 \\ 0.4 & 1 \\ 1 & -2 \end{bmatrix}$.

5.2.5 Simulation results

The response of this network is shown in Figure 5.5, for both state variables of all three subsystems. It is visible that the decentralised controllers are able to stabilise this system.

The structure of this network, as generated by Dinsdale, is shown in Figure 5.4.

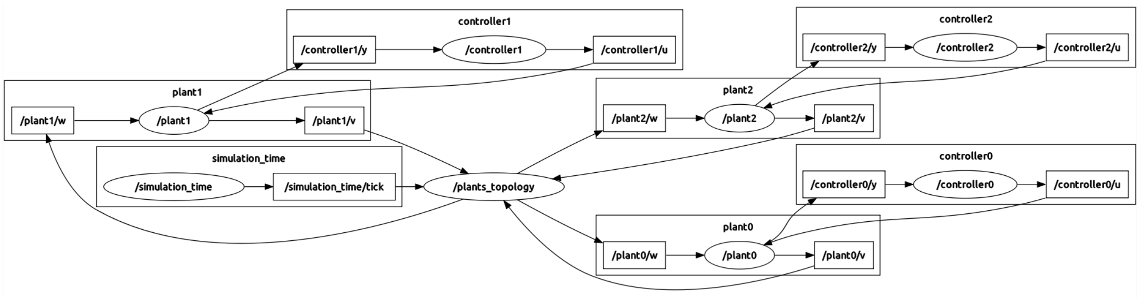
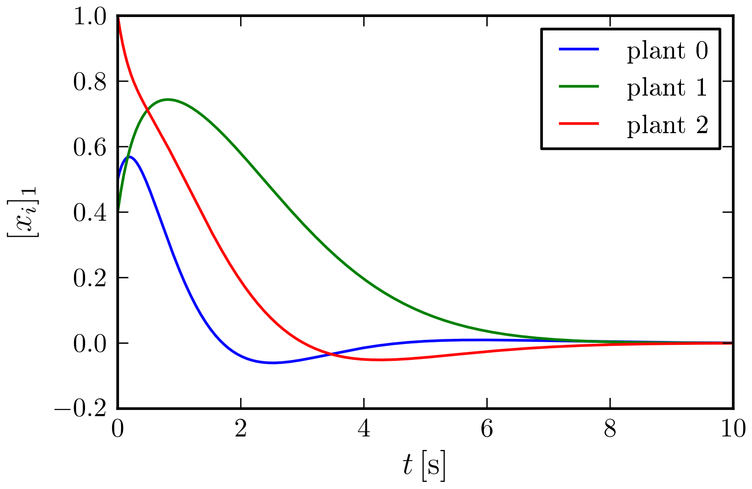
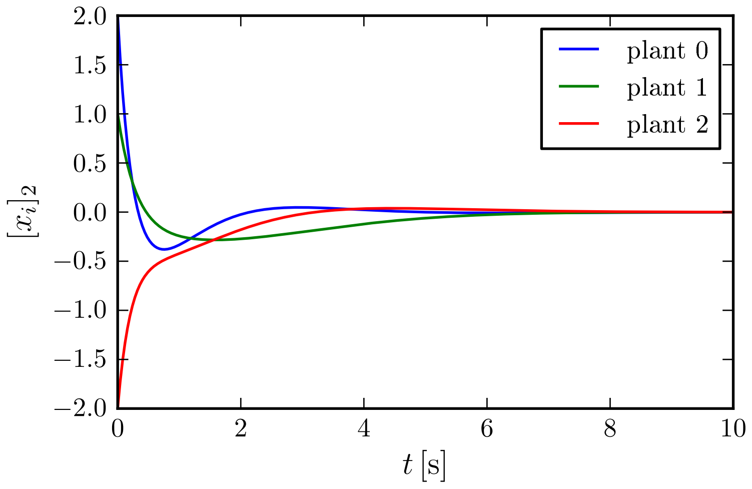


Figure 5.4: Runtime graph



(a) First state variable



(b) Second state variable

Figure 5.5: System response

5.3 Fleet formation fuzzy control with obstacle avoidance

This example will illustrate a network of physically decoupled dynamical systems. The subnetwork of plants (boats) is homogeneous, while the subnetwork of controllers is heterogeneous.

5.3.1 System description and control objectives

The network consists of a group of boats, controlled by decentralised control laws. The boats are not physically connected, but rather through measurements of mutual distance.

The objective in this example is to lead a fleet to a desired destination, while avoiding obstacles and maintaining a linear formation. This means that it is necessary to design such controllers which are going to be capable of maintaining a constant distance from the boat in front, while avoiding collision with obstacles. The controller on the first boat has to bring the vessel to a fixed target, rather than to follow another boat. The controllers must be designed in such a way that it will be trivial to add or remove boats.

This example is taken from [35], where in-depth analysis can be found.

5.3.2 Mathematical problem formulation

Assuming that the boat is moving in two dimensions, its kinematics can be written as:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ r \end{bmatrix}, \tag{13}$$

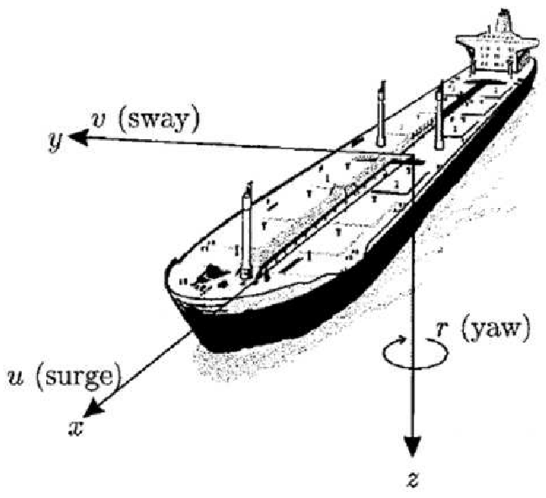


Figure 5.6: Degrees of freedom of a boat in 2D

where $(x,\, y)$ is the location of the boat's center of mass in the inertial frame, and ψ is it's orientation. All three together represent the vessel's position.

Under the assumption that environment forces due to winds, currents, and waves can be neglected, and that the inertia, added mass, and dumping matrices are diagonal, the dynamics of the vessel can be written as:

$$\begin{cases} \dot{u} &= \dfrac{m_{22}}{m_{11}}vr - \dfrac{d_{11}}{m_{11}}u + \dfrac{1}{m_{11}}\tau_1 \\ \dot{v} &= -\dfrac{m_{11}}{m_{22}}ur - \dfrac{d_{22}}{m_{22}}v \\ \dot{r} &= \dfrac{m_{11}-m_{22}}{m_{33}}uv - \dfrac{d_{33}}{m_{33}}r + \dfrac{1}{m_{33}}\tau_2 \end{cases}, \tag{14}$$

where m_{ii} are given by the vessel's inertia and added mass effect, and d_{ii} are defined by the hydrodynamic dumping. τ_1 and τ_2 are the control inputs: the force along the boat's longitudinal axis and the torque about its vertical axis.

To be able to control each boat, it is necessary to design a path planning algorithm, and a controller that will be able to keep the vessel on the computed path.

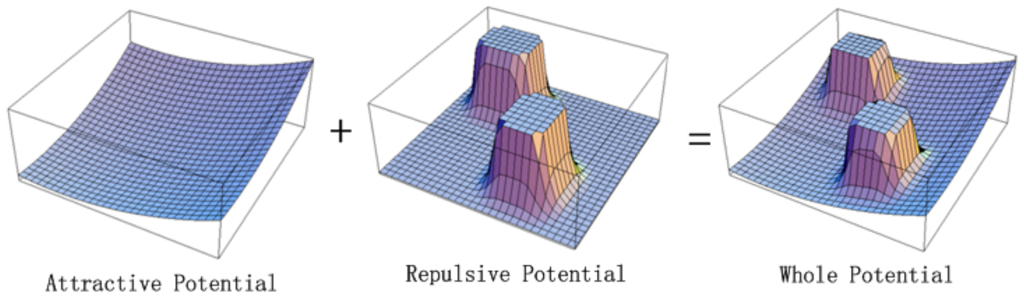


Figure 5.7: Potential fields

5.3.3 Control synthesis

As described in [35], the reference path of a boat is generated using the potential fields method. The principle is to generate a virtual potential function with the global minimum in the target's locations and maxima where the obstacles are. A simple example is shown in Figure 5.7. Once the field is generated, the path is calculated running a gradient descent algorithm on it. To avoid getting stuck in local minima, a rotational field around obstacles is also added to the standard attractive/repulsive one. The fields are computed locally, each controller generates its own field based on its target.

The attractive potential is computed as follows:

$$U_a = \frac{1}{4}\left((x-x_a)^2 + (y-y_a)^2 - R^2\right)^2, \tag{15}$$

where (x_a, y_a) is the location of the target, and R is the distance to maintain from it. In this case the target is either the ship being followed, or, in the case of the first boat, some predefined coordinates to reach.

The repulsive potential of obstacles is defined as:

$$U_r = \mu \sum_{i=1}^{M} \exp\left(-\frac{1}{2\sigma_i}\left((x-x_{p_i})^2 + (y-y_{p_i})\right)^2\right), \tag{16}$$

where M is the number of obstacles, σ is a measure of their width, and (x_p, y_p) are their coordinates. μ is a measure of strength of the repulsive field.

Finally, taking into consideration practical limitations of the boat and thus limiting its maximum speed, the path is generated as follows:

$$\begin{cases} \dot{x}_{\text{ref}} &= x_{\text{ref}_M} \tanh\left(-k_a \dfrac{\partial U_a}{\partial x} \exp\left(-k_{ra}\left(\dfrac{\partial U_r}{\partial x}\right)^2\right) - k_r \dfrac{\partial U_r}{\partial x} + m \dfrac{\partial U_r}{\partial y}\right) \\ \dot{y}_{\text{ref}} &= y_{\text{ref}_M} \tanh\left(-k_a \dfrac{\partial U_a}{\partial y} \exp\left(-k_{ra}\left(\dfrac{\partial U_r}{\partial y}\right)^2\right) - k_r \dfrac{\partial U_r}{\partial y} + m \dfrac{\partial U_r}{\partial x}\right) \end{cases}, \quad (17)$$

where x_{ref_M} and y_{ref_M} are constants used for limiting the speed.

To follow the generated path, an analytical fuzzy controller will be used. The control laws are designed accordingly to the following rules. The forward force should be proportional with the distance from the target, but with a certain upper limit. It should also decrease if the orientation error is significantly big, to prevent steering at high speeds. The steering torque, on the other hand, should be proportional to the orientation error, but also limited with an upper bound. From this description it is possible to formulate the following control laws:

$$\begin{cases} \tau_1 &= \tau_{1_M} \cdot \tanh\left(k_{11} \cdot no\right) \cdot \exp\left(-k_{21} \cdot vp^2\right) \\ \tau_2 &= \tau_{2_M} \cdot \tanh\left(k_{22} \cdot vp\right) \end{cases}, \quad (18)$$

where τ_{1_M} and τ_{2_M} are the upper bounds of force and torque. Here $no = \|\vec{r}\|$ represents the distance from the boat to the path to follow, and $vp = \dfrac{\vec{r}_e \times \vec{r}}{\|\vec{r}\|}$ the orientation error. The used vectors are shown in Figure 5.8.

5.3.4 Implementation

In this example the plant network is homogeneous – all boats are identical. The controller network however is heterogeneous, because the controller of the first boat is slightly different from the rest, which was done to show how to implement such combinations of networks.

The second thing for which this example is specific is its distributed control structure. The controllers exchange information on their position: each one receives the

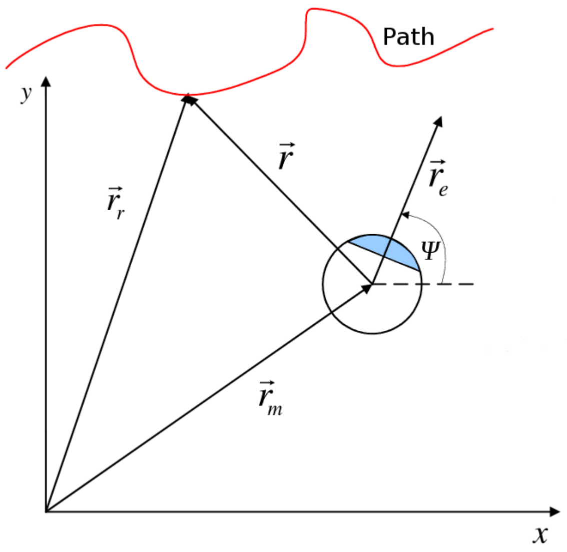


Figure 5.8: Vectors describing the boat's position

position of the boat it has to follow, from the respective controller. This is implemented in such way to demonstrate how to work with networks with distributed structures. The more natural and realistic way of implementing this network would be a decentralised control structure, where sensors to measure distance from other boats are implemented in the plant, and then the measurements are sent to the controller. Such implementation would be more suitable for practical applications, because only relative distances are used, without the need of knowing absolute positions, and also because it does not require the boats to be able to communicate with each other.

The input for the `__init__()`, `iterate_state()`, and `update_output()` methods of the `Plant` class inside `plant.py`, which describe the dynamics of a boat, are given in Listing 5.18, Listing 5.19, and Listing 5.20 respectively.

Listing 5.18: Boat initialisation

```
self.m = [200, 250, 80]
self.d = [70, 100, 50]
```

Listing 5.19: Boat state update

```
(x, y, psi, u, v, r) = np.nditer(self.x)

(u, v, r) = (self.T*(v*r*self.m[1]/self.m[0] - u*self.d[0]/self.m[0] +
    self.u[0, 0]/self.m[0]) + u,
             self.T*(-u*r*self.m[0]/self.m[1] - v*self.d[1]/self.m[1]) + v,
             self.T*(u*v*(self.m[0] - self.m[1])/self.m[2] -
                 r*self.d[2]/self.m[2] + self.u[1, 0]/self.m[2]) + r)
(x, y, psi) = (self.T*(u*np.cos(psi) - v*np.sin(psi)) + x,
               self.T*(u*np.sin(psi) + v*np.cos(psi)) + y,
               self.T*r + psi)

self.x = np.matrix([[x, y, psi, u, v, r]]).T
self.v = npm.zeros((2, 1))
```

Listing 5.20: Boat output update

```
self.y = self.x[:3, :]
```

The definition of the controller for the boats in the line (`controller.py`) is given in Listing 5.21 and Listing 5.22. Here, the list `self.o` contains information about obstacles: $[x, y, \sigma]$ for each obstacle. It is assumed that each boat knows where the obstacles are, rather than measuring them. The relative distance the boats have to keep from each other is set by `self.R`. Vector `self.p` is the output for neighbours (in this case the boat behind), while `self.q` is the input from neighbours (the boat in front).

The controller on the first boat (`controller_1.py`) is initialised using the same parameters as the other controllers (Listing 5.21), but it has slightly different equations in the update part, which are given in Listing 5.23.

Listing 5.21: Controller 0 and 1 initialisation

```
self.u_max = [200, 100]
self.k = [.5, 3, 10]
self.t = 0
self.k_p = [.2, 20, .5]
self.m = .1
self.ref_M = 1.5
self.R = 5
self.ref = npm.zeros((2, 1))
self.o = [[90, 0, 40],
          [35, 5, 50],
          [25, -30, 25],
         ]
```

Listing 5.22: Controller 0 update

```
if not self.t:
    self.ref = self.y[:2, :]
    self.q = self.ref
    self.t = 1

(x, y, psi) = np.nditer(self.y)
(x_ref, y_ref) = np.nditer(self.ref)
(x_q, y_q) = np.nditer(self.q)

r = (x_ref - x_q)**2 + (y_ref - y_q)**2 - self.R**2
F_a = [-r*(x_ref - x_q), -r*(y_ref - y_q)]
temp = [0, 0]
for obs in self.o:
    temp[0] -= (x_ref  - obs[0])*np.exp(-1/(2*obs[2])*((x_ref  - obs[0])**2 +
        (y_ref  - obs[1])**2))
    temp[1] -= (y_ref  - obs[1])*np.exp(-1/(2*obs[2])*((x_ref  - obs[0])**2 +
        (y_ref  - obs[1])**2))
F_r = []
F_r.append(self.k_p[1]*temp[0])
F_r.append(self.k_p[1]*temp[1])

(x_ref, y_ref) = (
    self.T*self.ref_M*np.tanh(.5*(F_a[0]*np.exp(-self.k_p[2]*F_r[0]**2) -
        F_r[0] + self.m*F_r[1])) + x_ref,
    self.T*self.ref_M*np.tanh(.5*(F_a[1]*np.exp(-self.k_p[2]*F_r[1]**2) -
        F_r[1] - self.m*F_r[0])) + y_ref)

no = np.sqrt((x_ref - x)**2 + (y_ref - y)**2)
vp = (y_ref - y)*np.cos(psi) - (x_ref - x)*np.sin(psi)
if no:
    vp /= no

self.u =
    np.matrix([[self.u_max[0]*np.tanh(self.k[0]*no)*np.exp(-self.k[1]*vp**2)],
                     [self.u_max[1]*np.tanh(self.k[2]*vp)]])
```

```
self.p = self.y[:2, :]
self.ref = np.matrix([[x_ref, y_ref]]).T
```

Listing 5.23: Controller 1 update

```
if not self.t:
    self.ref = self.y[:2, :]
    self.q = self.ref
    self.t = 1

(x, y, psi) = np.nditer(self.y)
(x_ref, y_ref) = np.nditer(self.ref)
(x_q, y_q) = 150, 0

F_a = [self.k_p[0]*(x_ref - x_q), self.k_p[0]*(y_ref - y_q)]
temp = [0, 0]
for obs in self.o:
    temp[0] -= (x_ref  - obs[0])*np.exp(-1/(2*obs[2])*((x_ref  - obs[0])**2 +
        (y_ref  - obs[1])**2))
    temp[1] -= (y_ref  - obs[1])*np.exp(-1/(2*obs[2])*((x_ref  - obs[0])**2 +
        (y_ref  - obs[1])**2))
F_r = []
F_r.append(self.k_p[1]*temp[0])
F_r.append(self.k_p[1]*temp[1])

(x_ref, y_ref) = (
    self.T*self.ref_M*np.tanh(.5*(-F_a[0]*np.exp(-self.k_p[2]*F_r[0]**2) -
        F_r[0] + self.m*F_r[1])) + x_ref,
    self.T*self.ref_M*np.tanh(.5*(-F_a[1]*np.exp(-self.k_p[2]*F_r[1]**2) -
        F_r[1] - self.m*F_r[0])) + y_ref
    )

no = np.sqrt((x_ref - x)**2 + (y_ref - y)**2)
vp = (y_ref - y)*np.cos(psi) - (x_ref - x)*np.sin(psi)
if no:
    vp /= no

self.u =
    np.matrix([[self.u_max[0]*np.tanh(self.k[0]*no)*np.exp(-self.k[1]*vp**2)],
                    [self.u_max[1]*np.tanh(self.k[2]*vp)]])
self.p = self.y[:2, :]
self.ref = np.matrix([[x_ref, y_ref]]).T
```

The parameters from `./input_parameters` used in this simulation are:

- `A_controllers` – Each controller is communicating its position to the one behind, therefore this adjacency matrix is $\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$.
- `A_plants` – The boat are not interacting in any way, however, this adjacency matrix always has to be a quadratic matrix the size of the number of nodes, which means in this case it will be populated with zeros: $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$.
- `controllers_iterate` – Controllers do communicate, but only once within a sampling period, not iteratively, therefore this is set to 0.
- `system_types` – All boats are the same, but the first one uses a different controller from the rest. This is described as $\begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$.
- `T` – This simulation uses a sampling period of 0.1 s. The real duration of a simulation step is set to 0.01 s, to speed up the simulation. The final time is chosen to be 115 s, which is just enough for the boats to reach the chosen target. This vector therefore is $\begin{bmatrix} 0.1 \\ 0.01 \\ 115 \end{bmatrix}$.
- `x0` – The initial conditions in this example are $\begin{bmatrix} 0 & -30 & 0 & 1 & 0 & 0 \\ 50 & 60 & -1.57 & 1 & 0 & 0 \\ 0 & 20 & 0.78 & 0 & 0 & 0 \end{bmatrix}$.

Accordingly to the requirement set in subsubsection 5.3.1, this system is designed in a way that it trivial to add or remove subsystems. To do so, it is only necessary to properly adjust the matrices in `./input_parameters`.

5.3.5 Simulation results

Figure 5.9 shows the trajectories of the three boats. The black circles represent obstacles, while the magenta circle is the target. It can be seen that, while the goal of the boats is to follow each other at a fixed distance, the priority is given to obstacle avoidance. This means that the formation can be achieved even in the middle of an obstacle field, like in the given example.

Figure 5.10 displays the graph generated by Dinsdale to simulate this system. It is interesting to note the topics `~/p`, which are used for communication between controllers, and are connected accordingly to `A_controllers`.

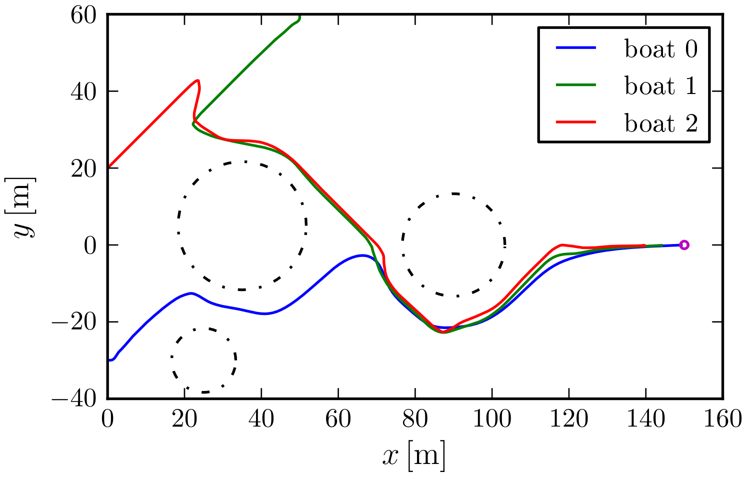


Figure 5.9: Boat trajectories

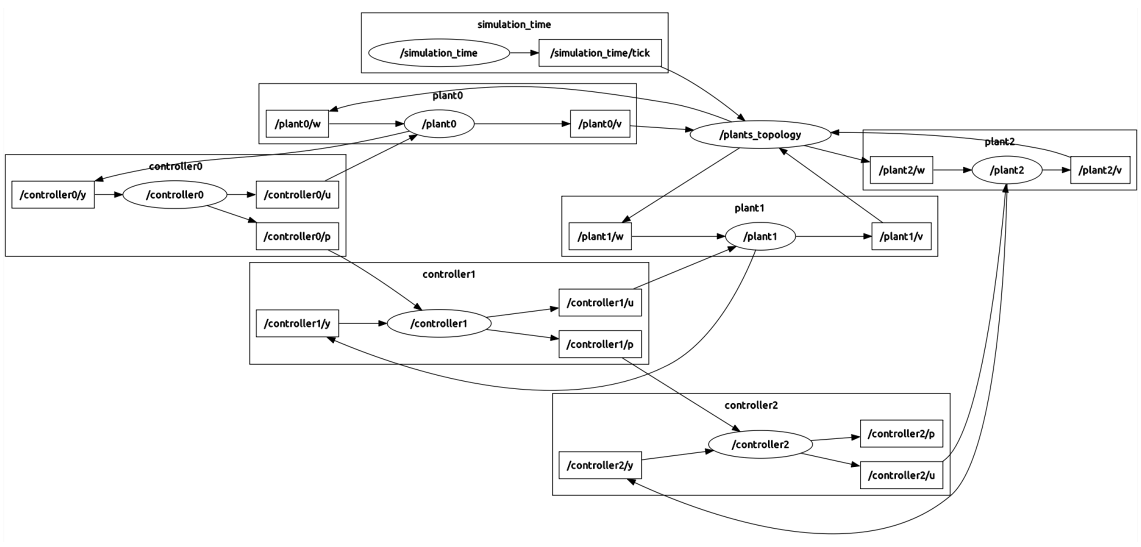


Figure 5.10: Runtime graph

5.4 Distributed optimisation problem

This example presents a network of two physically coupled dynamical subsystems, with a distributed control structure. The control law of each controller is not predefined, but instead it is computed online, solving an optimisation problem inside every sampling period, which requires controllers to collaborate.

5.4.1 System description and control objectives

This is an example presented in [36], and it deals with an unstable nonlinear system composed of two physically coupled mutually different subsystems, with a distributed control structure. The objective is to stabilise the entire system in an optimal manner. To achieve this, the controllers have to be connected in a distributed structure, for they need to collaborate in order to periodically solve a global optimisation problem in a distributed way. Not only the controllers have to communicate with each other, but they have to do it iteratively inside each sampling period.

5.4.2 Mathematical problem formulation

The discrete time dynamics of the interconnected systems are given in [36] as:

$$\begin{aligned} x_1(k+1) &= \begin{bmatrix} 1 & 0.7 \\ 0 & 1 \end{bmatrix} x_1(k) + \begin{bmatrix} \sin([x_1]_2) \\ 0 \end{bmatrix} + \begin{bmatrix} 0.245 \\ 0.7 \end{bmatrix} u_1(k) + \begin{bmatrix} 0 \\ ([x_2]_1)^2 \end{bmatrix} \\ x_2(k+1) &= \begin{bmatrix} 1 & 0.5 \\ 0 & 1 \end{bmatrix} x_2(k) + \begin{bmatrix} \sin([x_2]_2) \\ 0 \end{bmatrix} + \begin{bmatrix} 0.125 \\ 0.5 \end{bmatrix} u_2(k) + \begin{bmatrix} 0 \\ [x_1]_1 \end{bmatrix} \end{aligned}, \tag{19}$$

where $[\cdot]_i$ stands for the i-th element of a vector.

In order to achieve optimal stabilisation of this system, it is necessary to formulate a convex optimisation problem [37] that can be solved in a distributed way.

5.4.3 Control synthesis

In [36] the authors use a global linear Lyapunov function $V(x) = \|Px\|_\infty$ which is chosen using the technique presented in [38] applied to the system linearised around the origin. Here, the block diagonal matrix $P = \text{diag}(P_1, P_2)$ is computed using the method presented in [39], which yields:

$$\begin{aligned} P_1 &= \begin{bmatrix} 2.5598 & 0.3345 \\ 1.8629 & 5.0219 \end{bmatrix} \\ P_2 &= \begin{bmatrix} -0.3898 & -0.3836 \\ 0.2703 & 0.9763 \end{bmatrix} \end{aligned}. \tag{20}$$

The local constraint for each Lyapunov function is:

$$V_i(x_i(k+1)) - V_i(x_i(k)) \leq \tau_i(k) - \alpha_3^i(\|x_i(k)\|), \tag{21}$$

while the global constraint, which guarantees stability of the entire network, is:

$$\sum_i \tau_i \leq 0. \tag{22}$$

In addition to the global condition (22), each controller has to solve the local optimisation problem:

$$\begin{aligned} \min_{u_i} \quad & J_i(x_i(k+1), u_i(k)), \\ \text{subject to} \quad & x_i(k+1) = f_i(x_i(k), w_i(k), u_i(k)), \\ & V_i(x_i(k+1)) - V_i(x_i(k)) \leq \tau_i(k) - \alpha_3^i(\|x_i(k)\|), \end{aligned} \tag{23}$$

where J_i is a convex cost function, and w_i is the influence of other subsystems to subsystem i, which in this case is the last term of equations (19).

To satisfy condition (22), the controllers must exchange information. One way would be to have a central coordinator, which is connected to all controllers, and calculates this function. However, it is possible to show that the condition can be satisfied also with the controllers connected in a distributed structure, as long as the communication graph they form is connected. If this is the case, it follows that the global optimisation problem $\sum_i J_i$ will also be solved, which means that it is possible to obtain globally optimal behaviour.

5.4.4 Implementation

This example consist of a homogeneous network of two subsystems. The plants are significantly different, but the controllers use identical control laws. However, since model predictive control is used, it is necessary for the controller to have a model of the plant it is controlling. Therefore, controllers have to be implemented differently for different plants.

The implementation of the dynamics of the two plants, in `plant.py` and `plant_1.py`, and their respective controllers, are described in Table 5.2. It is extremely important to note that the convex optimisation solver cvxopt [40] is a requirement for running the controllers, and it should be imported at the beginning of each controller file as shown in Listing 5.29.

Table 5.2: Optimisation network definition

(a) Plants

Plant	`__init__()`	`iterate_state()`	`update_output()`
`plant.py`	Listing 5.24	Listing 5.25	Listing 5.26
`plant_1.py`	Listing 5.27	Listing 5.28	Listing 5.26

(b) Controllers

Controller	`__init__()`	`iterate_state()`	`iterate_optimisation()`
`controller.py`	Listing 5.30	Listing 5.31	Listing 5.32
`controller_1.py`	Listing 5.33	Listing 5.34	Listing 5.32

What is specific in this example is the use of iterative communication within a sampling period. The code which is executed in every such iteration goes inside the `iterate_optimisation()` module of the `Controller` class. The vector `self.r` is used to send data to neighbours, while `self.s` is where the data from neighbours is received. When the variable `self.finished` is set to `True`, the iterative communication stops and the normal execution flow continues. In the next time step, this variable will automatically be reset to `False`.

Listing 5.24: Plant 0 initialisation

```
Ts = .7
self.A = np.matrix([[1, Ts], [0, 1]])
self.B = np.matrix([[Ts**2/2.0], [Ts]])
```

Listing 5.25: Plant 0 state update

```
ff = np.matrix([[np.sin(self.x[1,0]), 0]]).T
vv = np.matrix([[0, self.w[0,0]**2]]).T
self.x = self.A*self.x + self.B*self.u + vv + ff
self.v = self.x[0,0]
```

Listing 5.26: Plant 0 output update

```
self.y = np.bmat([[self.x], [self.w[0]]])
```

Listing 5.27: Plant 1 initialisation

```
Ts = .5
self.A = np.matrix([[1, Ts], [0, 1]])
self.B = np.matrix([[Ts**2/2.0], [Ts]])
```

Listing 5.28: Plant 1 state update

```
ff = np.matrix([[np.sin(self.x[1,0]), 0]]).T
vv = np.matrix([[0, self.w[0,0]]]).T
self.x = self.A*self.x + self.B*self.u + vv + ff
self.v = self.x[0,0]
```

Listing 5.29: Importing cvxopt

```
from cvxopt import matrix, solvers
```

Listing 5.30: Controller 0 initialisation

```
solvers.options['show_progress'] = False

Ts = .7
self.A = np.matrix([[1, Ts], [0, 1]])
self.B = np.matrix([[Ts**2/2.0], [Ts]])

self.Qx = np.matrix('.1 0; 0 .1')
self.Px = np.matrix('4 0; 0 4')
self.Ru = .4

self.Q = np.matrix('.01 0; 0 .01')

self.P = np.matrix('2.5598 .3345; 1.8629 5.0219')

cx = 5

self.bxlow = -cx
self.bxup = cx

self.buup = 20

self.ALPHAconst = 2.0
```

Listing 5.31: Controller 0 update

```
self.Iter = 0
self.LAM = .1

self.tau = 1

self.f = np.matrix([[np.sin(self.y[1,0]), 0]]).T
self.v = np.matrix([[0, self.y[2,0]**2]]).T

self.Ju = self.B
self.Jx = self.A*self.y[:2,0] + self.f + self.v

self.Zu = self.Px*self.B
self.Zx = self.Px*self.Jx

self.Ku = self.P*self.B
self.Kx = self.P*self.Jx

self.c1 = npl.norm(self.Qx*self.y[:2,0], np.inf)
self.c2 = npl.norm(self.P*self.y[:2,0], np.inf) -
    npl.norm(self.Q*self.y[:2,0], np.inf)

self.u = np.matrix([[0]])
self.p = np.matrix([[0]])
```

Listing 5.32: Controller 0 and 1 optimisation iteration

```
if self.Iter:
    self.LAM += self.ALPHA*(self.tau + self.s[0,0])
    if ((abs(self.tau + self.s[0,0]) <= 1E-5 and abs(self.LAM*(self.tau +
        self.s[0,0])) < 1E-5) or self.Iter >= 100):
        self.finished = True
        return

self.Iter += 1
self.ALPHA = self.ALPHAconst/np.sqrt(self.Iter)

fL = np.matrix([[1, 1, 0, self.LAM]]).T

AL1 = np.matrix([[0, 0, 1, 0],
                 [0, 0, -1, 0],
                 [-1, 0, 0, 0],
                 [0, -1, 0, 0],
                 [0, 0, 0, 1],
                 [0, 0, 0, -1],
                 [0, 0, self.Ju[0,0], 0],
                 [0, 0, -self.Ju[0,0], 0],
                 [0, 0, self.Ju[1,0], 0],
                 [0, 0, -self.Ju[1,0], 0],
                 [0, 0, self.Ku[0,0], -1],
                 [0, 0, -self.Ku[0,0], -1],
                 [0, 0, self.Ku[1,0], -1],
                 [0, 0, -self.Ku[1,0], -1],
                 [-1, 0, self.Zu[0,0], 0],
                 [-1, 0, -self.Zu[0,0], 0],
                 [-1, 0, self.Zu[1,0], 0],
                 [-1, 0, -self.Zu[1,0], 0],
                 [0, -1, self.Ru, 0],
                 [0, -1, -self.Ru, 0]])

bL1 = np.matrix([[self.buup],
                 [self.buup],
                 [0],
                 [0],
                 [1000],
                 [1000],
                 [self.bxup - self.Jx[0,0]],
                 [self.Jx[0,0] - self.bxlow],
                 [self.bxup - self.Jx[1,0]],
                 [self.Jx[1,0] - self.bxlow],
                 [-self.Kx[0,0] + self.c2],
                 [self.Kx[0,0] + self.c2],
                 [-self.Kx[1,0] + self.c2],
                 [self.Kx[1,0] + self.c2],
                 [-self.c1 - self.Zx[0,0]],
                 [-self.c1 + self.Zx[0,0]],
                 [-self.c1 - self.Zx[1,0]],
                 [-self.c1 + self.Zx[1,0]],
                 [0],
                 [0]])

AL_c = matrix(AL1)
bL_c = matrix(bL1)
fL_c = matrix(fL)

sol  = solvers.lp(fL_c, AL_c, bL_c)
Opt = sol['x']

self.u = np.matrix([[Opt[2,0]]])
self.tau = Opt[3,0]

self.r = np.matrix([[self.tau]])
```

Listing 5.33: Controller 1 initialisation

```
solvers.options['show_progress'] = False

Ts = .5
self.A = np.matrix([[1, Ts], [0, 1]])
self.B = np.matrix([[Ts**2/2.0], [Ts]])

self.Qx = np.matrix('.1 0; 0 .1')
self.Px = np.matrix('4 0; 0 4')
self.Ru = .4

self.Q = np.matrix('.01 0; 0 .01')

self.P = np.matrix('-.3898 -.3836; .2703 .9763')

cx = 5

self.bxlow = -cx
self.bxup = cx

self.buup = 20

self.ALPHAconst = 2.0
```

Listing 5.34: Controller 1 update

```
self.Iter = 0
self.LAM = .1

self.tau = 1

self.f = np.matrix([[np.sin(self.y[1,0]), 0]]).T
self.v = np.matrix([[0, self.y[2,0]]]).T

self.Ju = self.B
self.Jx = self.A*self.y[:2,0] + self.f + self.v

self.Zu = self.Px*self.B
self.Zx = self.Px*self.Jx
self.Ku = self.P*self.B
self.Kx = self.P*self.Jx

self.c1 = npl.norm(self.Qx*self.y[:2,0], np.inf)
self.c2 = npl.norm(self.P*self.y[:2,0], np.inf) -
    npl.norm(self.Q*self.y[:2,0], np.inf)

self.u = np.matrix([[0]])
self.p = np.matrix([[0]])
```

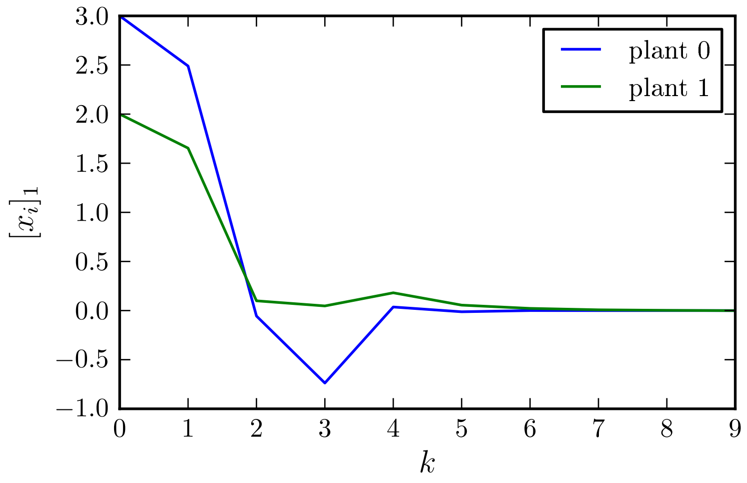
The parameters in `./input_parameters` used for this simulation are:

- `A_controllers` – The adjacency matrix of the controllers is $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.
- `A_plants` – The plants interaction network has the same topology as the controller communication network, which is $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.
- `controllers_iterate` – Controllers communicate iteratively, therefore this is set to 1.
- `system_types` – The network is homogeneous: $\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$.
- `T` – This simulation uses an iteration step (sampling period) of size 1. The real duration of a simulation step is set to 0.5 s, to guarantee enough time for iterative communication and iterative linear program solving. The final number of steps (final time) is chosen to be 10, which is enough for the controllers to stabilise the network. This vector therefore is $\begin{bmatrix} 1 \\ 0.5 \\ 10 \end{bmatrix}$.
- `x0` – The initial conditions chosen in this example are $\begin{bmatrix} 3 & 2 \\ 2 & 0.5 \end{bmatrix}$.

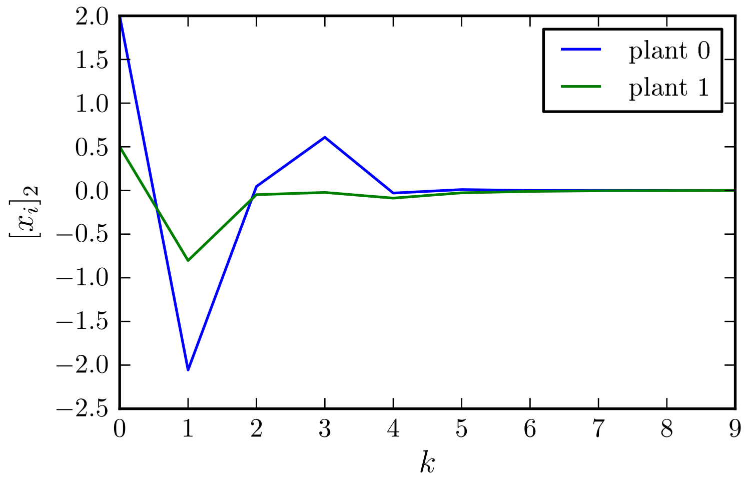
5.4.5 Simulation results

Figure 5.11 shows the response of the network. The controllers are able to stabilise the system in 4 to 5 steps. Within each step there is up to a hundred iterations of optimisation problem solving.

In Figure 5.12 it is possible to see the graph generated by Dinsdale to simulate this



(a) First state variable



(b) Second state variable

Figure 5.11: System response

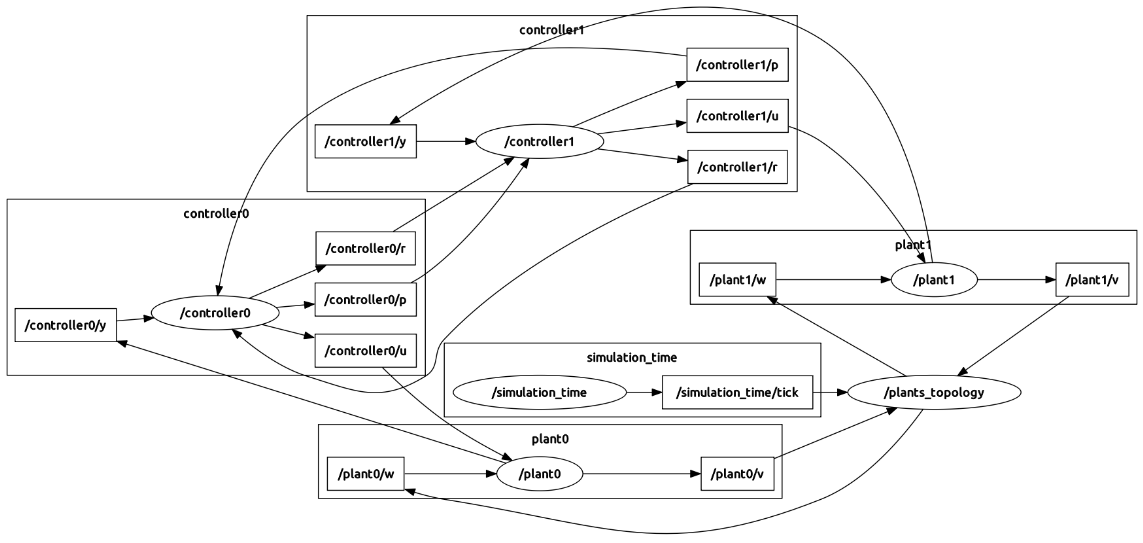


Figure 5.12: Runtime graph

network. It is interesting to note the `~/r` topics within the `controller*` namespaces. They are used for iterative communication between controllers, and their connection topology is the same as that of `~/p` topics.

6 Conclusion and future developments

This thesis presented the motivation and explored the possibilities of creating a universal framework for development of networks of dynamical systems. It introduced a new framework, which fully satisfies the requirements, derived by the needs of today's research community. With the testing conducted so far, it has been shown that the Dinsdale framework offers a solid and reliable infrastructure for research and development of dynamical networks. Although the benefits from using such a framework are theoretically clear, they still have to be proven in practice. Feedback from other research groups would be immensely valuable for determining the direction of the future development of this software.

Generally speaking, the need for such framework exists, and considering its foundations are the increasingly popular ROS and Python, its adoption could be very convenient. It will certainly remain available and continue being maintained and developed, at least in the foreseeable future.

References

[1] R. Murray, K. Astrom, S. Boyd, R. Brockett, and G. Stein, "Future directions in control in an information-rich world," *Control Systems, IEEE*, vol. 23, pp. 20–33, Apr 2003.

[2] A. Preumont, *Vibration Control of Active Structures: An Introduction.* Solid Mechanics and Its Applications, Springer, 2011.

[3] P. Koumoutsakos and I. Mezic, *Control of Fluid Flow.* Lecture Notes in Control and Information Sciences, Springer, 2006.

[4] R. Duffner, *The Adaptive Optics Revolution: A History.* University of New Mexico Press, 2009.

[5] J. Ekanayake, N. Jenkins, K. Liyanage, J. Wu, and A. Yokoyama, *Smart Grid: Technology and Applications.* Wiley, 2012.

[6] H. S. Witsenhausen, "A counterexample in stochastic optimum control," *SIAM Journal on Control*, vol. 6, no. 1, pp. 131–147, 1968.

[7] R. Olfati-Saber, "Flocking for multi-agent dynamic systems: Algorithms and theory," *IEEE Transactions on Automatic Control*, vol. 51, pp. 401–420, 2006.

[8] M. Mesbahi and M. Egerstedt, *Graph Theoretic Methods in Multiagent Networks.* Princeton Series in Applied Mathematics, Princeton University Press, 2010.

[9] F. Bullo, J. Cortés, and S. Martínez, *Distributed Control of Robotic Networks.* Applied Mathematics Series, Princeton University Press, 2009. `http://coordinationbook.info`.

[10] G. Antonelli, "Interconnected dynamic systems: An overview on distributed control," *Control Systems, IEEE*, vol. 33, no. 1, pp. 76–88, 2013.

[11] M. Rotkowitz and S. Lall, "A characterization of convex problems in decentralized control," *Automatic Control, IEEE Transactions on*, vol. 51, pp. 274–286, Feb 2006.

[12] S. Jiang, P. Voulgaris, L. Holloway, and L. Thompson, "Distributed control of large segmented telescopes," in *American Control Conference, 2006*, pp. 6 pp.–, June 2006.

[13] V. D. Blondel and J. N. Tsitsiklis, "A survey of computational complexity results in systems and control," *Automatica*, vol. 36, no. 9, pp. 1249 – 1274, 2000.

[14] B. Bamieh, F. Paganini, and M. Dahleh, "Distributed control of spatially invariant systems," *Automatic Control, IEEE Transactions on*, vol. 47, pp. 1091–1107, Jul 2002.

[15] R. D'Andrea and G. Dullerud, "Distributed control design for spatially interconnected systems," *Automatic Control, IEEE Transactions on*, vol. 48, pp. 1478–1495, Sept 2003.

[16] B. Recht and R. D'Andrea, "Distributed control of systems over discrete groups," *Automatic Control, IEEE Transactions on*, vol. 49, pp. 1446–1452, Sept 2004.

[17] C. Langbort, R. Chandra, and R. D'Andrea, "Distributed control design for systems interconnected over an arbitrary graph," *Automatic Control, IEEE Transactions on*, vol. 49, pp. 1502–1519, Sept 2004.

[18] B. Bamieh and P. G. Voulgaris, "A convex characterization of distributed control problems in spatially invariant systems with communication constraints," *Systems & Control Letters*, vol. 54, no. 6, pp. 575 – 583, 2005.

[19] N. Motee and A. Jadbabaie, "Optimal control of spatially distributed systems," *Automatic Control, IEEE Transactions on*, vol. 53, pp. 1616–1629, Aug 2008.

[20] J. Rice and M. Verhaegen, "Distributed control: A sequentially semi-separable approach for spatially heterogeneous linear systems," *Automatic Control, IEEE Transactions on*, vol. 54, pp. 1270–1283, June 2009.

[21] C. Langbort, L. Xiao, R. D'Andrea, and S. Boyd, "A decomposition approach to distributed analysis of networked systems," in *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, vol. 4, pp. 3980–3985 Vol.4, Dec 2004.

[22] U. Jönsson, C.-Y. Kao, and H. Fujioka, "A popov criterion for networked systems," *Systems & Control Letters*, vol. 56, no. 9-10, pp. 603 – 610, 2007.

[23] S. Nersesov and M. Haddad, "On the stability and control of nonlinear dynamical systems via vector lyapunov functions," *Automatic Control, IEEE Transactions on*, vol. 51, pp. 203–215, Feb 2006.

[24] C. W. Scherer, "Structured finite-dimensional controller design by convex optimization," *Linear Algebra and its Applications*, vol. 351–352, no. 0, pp. 639 – 669, 2002. Fourth Special Issue on Linear Systems and Control.

[25] I. Lestas and G. Vinnicombe, "Scalable decentralized robust stability certificates for networks of interconnected heterogeneous dynamical systems," *Automatic Control, IEEE Transactions on*, vol. 51, pp. 1613–1625, Oct 2006.

[26] A. Gattami and R. Murray, "A frequency domain condition for stability of interconnected mimo systems," in *American Control Conference, 2004. Proceedings of the 2004*, vol. 4, pp. 3723–3728 vol.4, June 2004.

[27] E. Camponogara, D. Jia, B. Krogh, and S. Talukdar, "Distributed model predictive control," *Control Systems, IEEE*, vol. 22, pp. 44–52, Feb 2002.

[28] W. Dunbar, "Distributed receding horizon control of dynamically coupled nonlinear systems," *Automatic Control, IEEE Transactions on*, vol. 52, pp. 1249–1263, July 2007.

[29] B. T. Stewart, A. N. Venkat, J. B. Rawlings, S. J. Wright, and G. Pannocchia, "Cooperative distributed model predictive control," *Systems & Control Letters*, vol. 59, no. 8, pp. 460 – 469, 2010.

[30] P. D. Christofides, R. Scattolini, D. M. de la Peña, and J. Liu, "Distributed model predictive control: A tutorial review and future research directions," *Computers & Chemical Engineering*, vol. 51, no. 0, pp. 21 – 41, 2013. {CPC} {VIII}.

[31] D. Šiljak and D. Stipanović, "Robust stabilization of nonlinear systems: The LMI approach.," *Mathematical Problems in Engineering*, vol. 6, no. 5, pp. 461–493, 2000.

[32] D. Siljak, *Decentralized Control of Complex Systems.* Mathematics in science and engineering, Academic Press, 1991.

[33] H. Khalil, *Nonlinear Systems.* Prentice Hall PTR, 2002.

[34] S. Boyd, L. Ghaoul, E. Feron, and V. Balakrishnan, *Linear Matrix Inequalities in System and Control Theory.* Studies in Applied Mathematics, Society for Industrial and Applied Mathematics, 1994.

[35] M. Rossi, "Upravljanje flotom," tech. rep., University of Zagreb, 2014.

[36] A. Jokic and M. Lazar, "On stabilization of discrete-time nonlinear systems under arbitrary information constraints," in *1st IFAC Workshop on Estimation and Control of Networked Systems, Venice, Italy*, September 2009.

[37] S. Boyd and L. Vandenberghe, *Convex Optimization.* Cambridge University Press, 2004.

[38] A. Jokic and M. Lazar, "On decentralized stabilization of discrete-time nonlinear systems," in *American Control Conference, 2009. ACC '09.*, pp. 5777–5782, June 2009.

[39] M. Lazar, W. P. M. H. Heemels, S. Weiland, and A. Bemporad, "Stabilizing model predictive control of hybrid systems," *Automatic Control, IEEE Transactions on*, vol. 51, pp. 1813–1818, Nov 2006.

[40] M. Andersen, J. Dahl, and L. Vandenberghe, "cvxopt." `http://cvxopt.org/`.

[41] M. Rossi, "ROS u distribuiranom upravljanju dinamičkim sustavima," tech. rep., University of Zagreb, 2014.

[42] ROS, "Documentation." `http://wiki.ros.org/`.

[43] J. M. O'Kane, *A Gentle Introduction to ROS.* Independently published, Oct. 2013. Available at `http://www.cse.sc.edu/~jokane/agitr/`.

[44] ROS, "About ROS." `http://www.ros.org/about-ros/`.

[45] Willow Garage, "About Us." `http://www.willowgarage.com/pages/about-us`.

[46] OSRF, "ROS @ OSRF." `http://osrfoundation.org/blog/ros-at-osrf.html`.

[47] Robohub, "ROS 101: Intro to the Robot Operating System." `http://robohub.org/ros-101-intro-to-the-robot-operating-system/`.

[48] R. Diestel, *Graph Theory, 4th Edition*, vol. 173 of *Graduate texts in mathematics.* Springer, 2012.

[49] C. Godsil and G. Royle, *Algebraic Graph Theory.* Graduate Texts in Mathematics, Springer, 2001.

A Robot Operating System

This is a very brief overview of the main concepts of ROS, required to understand certain parts of this thesis. A more detailed introduction, with focus on distributed system development, can be found in [41]. More in-dept information on ROS, as well as tutorials, can be found online, on the ROS website [42], and in books, like [43].

ROS is a framework primarily intended for development of robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a variety of robotic platforms [44]. ROS is free software, licensed under the permissive BSD license. It was created, and, from 2007 to 2013, developed, and maintained mostly by Willow Garage, a research laboratory and technological incubator which produces hardware and free software for service robotics [45]. Since 2013 the project has been transfered to the Open Source Robotics Foundation (OSRF) [46], a non-profit organization born as a Willow Garage spin-off.

It is important to note that ROS is not a replacement for a computer operating system, but a middleware between the robot hardware and the control algorithms. From the robotics point of view however, it offers all the services and abstractions expected from an operating system, which justifies its name. Some of the key services are hardware abstraction, low level device control, implementation of commonly used functions, process management, and package management. It also provides tools and libraries for obtaining, building, writing and running code across multiple computers [42]. At the moment, ROS runs on top of UNIX-like operating systems, with the best support on the Ubuntu GNU/Linux distribution and its derivatives.

Software in ROS is distributed in packages and stacks of packages. A package is simply a directory with a certain structure, which can contain executables, libraries, configuration files, or anything else.

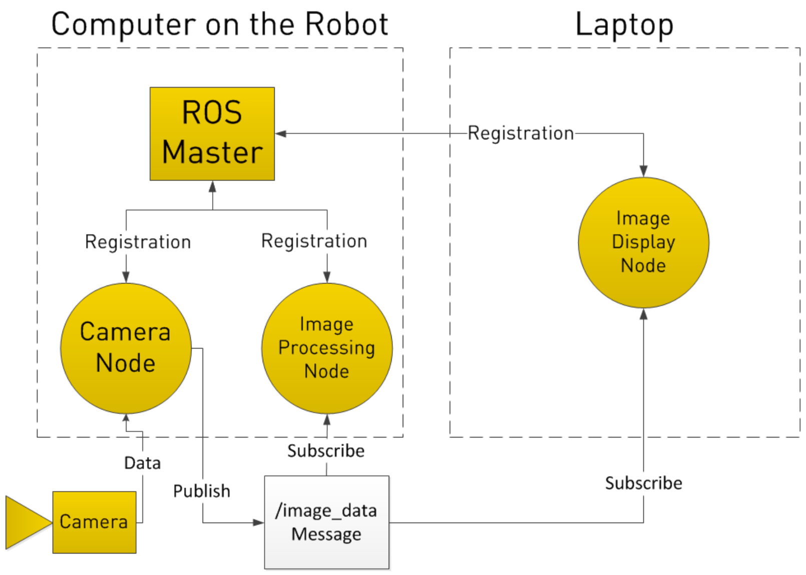
At runtime, ROS manages the software inside what is called the ROS Computational Graph, which is basically a network of processes that can communicate with each other. The main concepts are:

- Nodes
- Master
- Parameter Server
- Messages
- Topics
- Services
- Bags

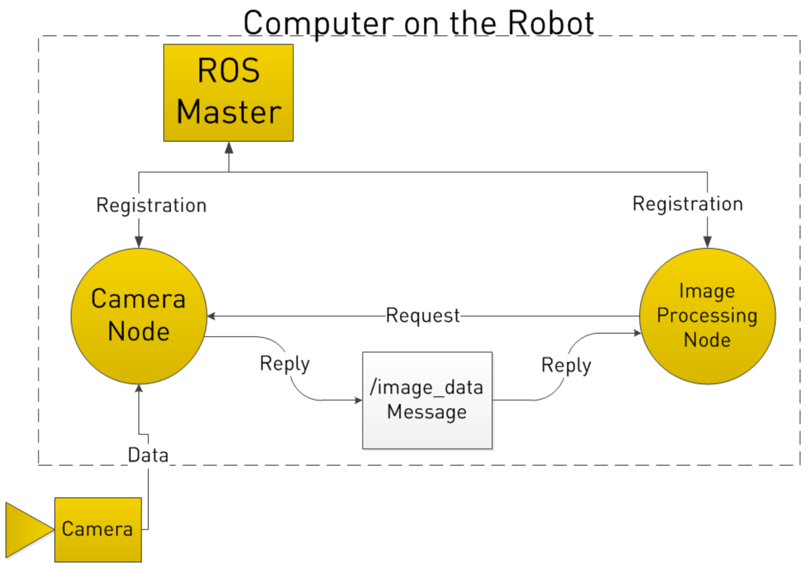
The programs (processes) that are being executed are called nodes, while the vertices of the graph are the interactions between them. The topology of the graph can be dynamical: the graph can be fully modified at runtime.

The main program (server) which allows the interaction between nodes, because it contains all the names and addresses, is called Master. The Parameter Server is a part of the Master which serves as a place for storing publicly available data in a centralised location. It is not very fast, so it is not used for runtime communication, but rather for storing some universal parameters.

There are two methods for communication between nodes: the asynchronous publishing of messages to topics, and the synchronous communication through services. A node can send a message by publishing it to a given topic. The topic is a name to identify the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There is no limit to the number of publishers



(a) Publishing and subscribing to a topic



(b) Using a service

Figure A.1: Examples of ROS systems [47]

and subscribers for a topic. Services are used in cases when request/reply communication is required. A node can offer a service under a specific name, to which any other node can send a request and wait for the reply.

The last fundamental concept that is important to understand are bags. Bags are files used for storage and reproduction of logged data. They offer a convenient way to store all kind of data during execution, which is very valuable for development and testing. The data from a bag file can be reproduced in the order it was collected, which allows for example testing different algorithms on the same data set.

Figure A.1 illustrates two simple examples of ROS graphs. Figure A.1a shows a ROS system that runs on two computers: one on the robot and one for the human interface. The ROS Master, where each node has to register to be visible to others, is running on the robot. Only three nodes are running: one that acquires images from the camera, one for image processing, and one to display the acquired images on the user's machine. In this case the camera node publishes the images on a topic, to which the other two are subscribed. In Figure A.1b there are only two nodes, and this time the camera node is offering a service. When the image processing node needs an image, it sends a request to the service and waits for the answer from the node offering the service.

B Graph terminology

Graph theory is useful when it is necessary to model the concepts of proximity and interactions among agents. For example, if the agents use limited range communication, like in Figure B.1, a graph could be the most appropriate tool to handle the concept of connectivity among them. Graph-based abstractions of networked systems contain virtually no information about what exactly is shared by the agents, the communication protocol, or what is subsequently done with the shared data. Instead of this, the graph-based abstraction is a high-level description of the network topology, using objects referred to as vertices and edges. This section will present the most basic concepts, while detailed information can be found in [48] and [49].

A directed graph, or digraph, is a pair $G = (V, E)$, which consists of a set of nodes (vertices) $V = \{1, 2, \ldots, n\}$, and a set of edges $E \subseteq \{(i, j) : i, j \in V, j \neq i\}$ (the graph does not contain self loops). A graph is said to be undirected if $(i, j) \in E \Leftrightarrow (j, i) \in E$. A weighted graph associates a weight to each edge. When an edge belongs to E, it means that there is information flow from the tail node i to the head node j. The quantities $|V|$ and $|E|$ (number of nodes and edges) are referred to as graph order and communication complexity respectively.

An appealing feature of graphs is the possibility to describe many of their features with matrices. Given a graph G, the adjacency matrix $A \in \mathbb{R}^{|V|\times|V|}$ can be defined as

$$A = [a_{ij}], \;\; a_{ij} = \begin{cases} 1, & j \in N_i \\ 0, & \text{otherwise} \end{cases} . \tag{24}$$

The graph is called weighted whenever the elements of its adjacency matrix are other than $\{0, 1\}$. The set of neighbours of node v_i can be defined as:

$$N_i = \{j \in V : a_{ij} \neq 0\} = \{j \in V : (i, j) \in E\}, \tag{25}$$

which defines the set of all nodes that node v_i can take information from. The

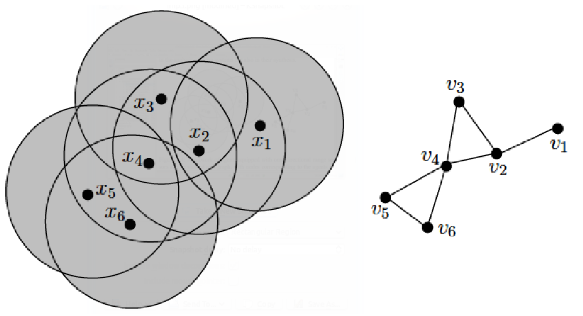


Figure B.1: Graph representation of nodes with limited interaction range [8]

number of such nodes $|N_i|$ is called the in-degree of a node. It is possible to define the diagonal input degree matrix $D \in \mathbb{R}^{|V|\times|V|}$ of the entire graph as

$$D = [d_{ij}], \ \ d_{ij} = \begin{cases} |N_i|, & i = j \\ 0, & \text{otherwise} \end{cases} . \tag{26}$$

The graph Laplacian matrix $L \in \mathbb{R}^{|V|\times|V|}$ is defined as

$$L = D - A = [l_{ij}], \tag{27}$$

which is equivalent to

$$l_{ij} = \begin{cases} -1, & j \in N_i \\ |N_i|, & i = j \\ 0, & \text{otherwise} \end{cases} . \tag{28}$$

Other important basic graph properties are related to connectivity. A graph is strongly connected when there exists a directed path connecting every arbitrary pair of distinct nodes. A graph is connected, or weakly connected, when there exists an undirected path connecting every arbitrary pair of distinct nodes. For undirected graphs the two definitions coincide.