Designing a generic software architecture for ARW manipulators

P.C. (Pieter) Dijkshoorn

BSc Report

Committee:
Dr.ir. J.F. Broenink
Dr.ir. R.G.K.M. Aarts
H.W. Wopereis, MSc

November 2016

046RAM2016
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

AEROWORKS is an international project, that aims to develop a team of autonomous maintenance and inspection drones, also named Aerial Robotic Workers (ARWs), to be deployed in the infrastructure sector. Environmental hazards may complicate these tasks when performed by humans and reduce the safety of the operation. The ARWs utilize manipulators (robotic arms) equiped with tools which allow them to perform a variety of tasks. These manipulators come in many forms and sizes, and so does the software used to control them.

The aim of this project is to generalize this software and make it reconfigurable to such an extend that virtually any manipulator is supported by it. This allows all AEROWORKS manipulators to be controlled using a single software and prevents the need to write a program for every newly developed manipulator.

Previously created software for one of the manipulators was analysed and decomposed, to find out what is the least required in order to control a manipulator. By making this most basic framework independent of the manipulator in use, and allowing it to be easily reconfigured, a software is created that acts as a shell, hosting generic functionality that can be adapted to the user's needs.

The final software that was created, was tested using several configurations, and performed as expected in each. Core functionality and manipulator specific settings were seperated, allowing the latter to be edited without having to deal with the first. These specific settings can be edited per component, reducing difficulty in making modifications. In order to drive motors other than polulu motors, the microcontroller might have to be reprogrammed or replaced.

Some recommendations for future work are given regarding optimizations and extensions that can be made to the software. These mainly deal with the fact that some specific assumptions have been made, that could have been generalized.

Contents

1 Introduction

1.1 Context

The AEROWORKS Aerial Robotic Workers are drones, unmanned aerial vehicles equipped with manipulators to perform tasks in hazardous or hard-to-reach environments. They can be deployed in order to ensure safety of a human operator, or can perform autonomous inspection and maintenance.

To perform maintenance tasks, Aerial Robotic Workers (ARWs) use manipulators, such as robotic arms equipped with end effectors. These arms come in different forms and sizes and in order to control them, slightly different software is required for each. To control these manipulators a generic, tunable software has been written so that not every single manipulator needs its own software. Although this software is reconfigurable to an extend, it is very restricted in what types of manipulators it supports. By improving this software it will become easier to apply to different manipulators, as well as reduce development and deployment times of new designs.

1.2 Goal

This project describes the design of a generic software architecture for manipulators used by aerial robotic workers (ARW's). The aim is to make the software as generic and reconfigurable as possible, such that it is applicable to a wide range of manipulators, while keeping it user-friendly in both the GUI and code-readability.

1.3 Approach

Since a working manipulator software is available initially, instead of designing the whole software from scratch, a close look will be taken at the initial software and adjustments will be made where necessary. In this process, key will be to determine what parts of the software are manipulator specific, and what parts can be shared between manipulators. After refactoring, the software will be clearly divided between core and specific components.

1.4 Report outline

In chapter two some background information on AEROWORKS, manipulators and ROS will be given. The initially used software is analyzed in chapter three, where improvement requirements are presented as well. Design choices and their implementation are discussed in chapter four and the final results are presented in chapter five. Conclusions and recommendations for future work are found in chapter six.

2 Background

2.1 AEROWORKS

AEROWORKS (2016b) is a collaboration project between six universities and four companies in Europe, aiming to develop a team of Aerial Robotic Workers to aid the inspection and maintenance of the growing infrastructure, in developing and developed countries. The participating universities work separately on specific aspects of the ARW's to create an optimal solution for the infrastructure sector. One of these aspects is the physical interaction of the ARW with its environment through various manipulators.

2.2 ROS

The Robotic Operating System (ROS) is a communication framework used by the robotic workers to send data back an forth between different parts of the system. It can be used to connect different programs, called nodes, which may be written in different languages. Nodes are connected using topics, on which messages can be published by these nodes. If a message is published on a topic, the nodes subscribed to said topic receive the message and the data it contains. Different components or functions in the system are given their own node, structuring the whole system in a convenient way. It allows developers of individual nodes to choose their way of implementation as long as their node fits in the whole.

Since within the AEROWORKS project, the different universities work on separate aspects of the whole system, ROS is a convenient structure to use. Another benefit of the modular nature of ROS is that in the event of failure, the failing node can easily be identified and fixed. Debugging can also be done using several command line tools build into ROS. These allow users to view topic activity and publish a message to any active topic.

Finally, a feature of ROS that is used quite often are services. These are function-like constructs that can be called by any node with specified arguments. They then return arguments, just like functions do, to the node that called the service. Usually services contain mathematical calculations that are long and preferably omitted in the node scripts (ROS.org, 2016).

2.3 Available manipulators

The manipulators used by the ARWs are made with the purpose of being mounted on a drone, and are thus lightweight arms. Several different manipulators designed by some of the contributing universities have already been tested with an ARW. ETH from Zürich has made a manipulator with three degrees of freedom (DOFs), x, y and theta, able to extend, shift right and left in an arc, and pitch up and down, seen in Figure 2.1. It uses three motors to accomplish this, one to control the pitch of the manipulator with respect to the ARW, and two to bend two side arms part of a scissor mechanism. As a result, the end-effector can be moved towards and away from the base, as well as shifted left and right.

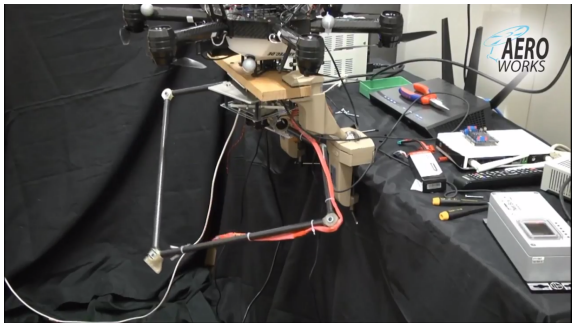

Figure 2.1: The manipulator made in Zürich (AEROWORKS, 2016a).

At the University of Twente, a hexagonal manipulator with three DOFs was prototyped, x, z, theta, see the photograph in Figure 2.2. One of the sides of the hexagon is connected parallel to the base of the ARW, the opposite side attaches to the end-effector. Three motors are used

to drive two joints connected to the base and one of the middle joints. Two of the joints are controlled through a differential gear, by two motors working together (Dekker, 2015).

By mounting this manipulator on its base (not shown in the picture), two more motors are required to add degrees of freedom in Y and yaw. This too is done using a differential gear.

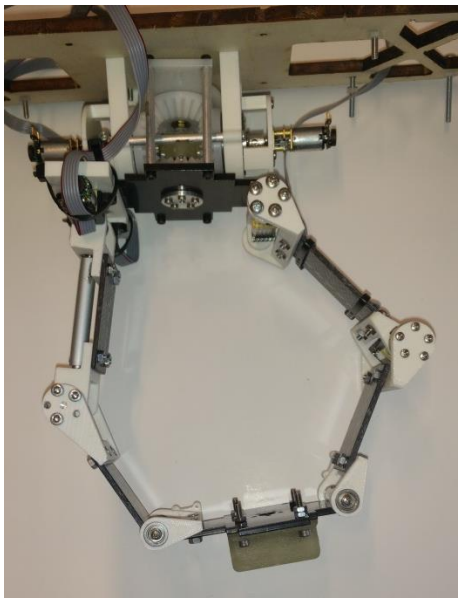

Figure 2.2: The hexagonal UT manipulator (Dekker, 2015).

Figure 2.3: LTU manipulator 'Carma' mounted under a drone (Kominiak, 2016).

The University of Luleå, LTU, has made a robotic arm consisting of four joints, connecting four serial links and the end-effector (Figure 2.3). It allows the end-effector to move with three DOFs, x, z and theta. To transfer rotation from the motors to the joints belts have been used, so that all motors can be positioned at the base. This manipulator can either use magnetic encoders on the motor axes to measure their angular velocity, or potentiometers to measure the positions of the belts and consequently the joint positions Wuthier (2016).

3 Analysis

In this chapter, the purpose of the software will be presented. A close look will be taken at the software in its state at the start of the project (Schipper, 2016) to identify which components can be used by all manipulators, and can be separated from components that change when switching manipulators. The original code will be reused wherever possible to save time. After identification of the core and specific components, software requirements are presented.

3.1 Purpose of the software

The purpose of the software is to control a wide variety of existing manipulators, or manipulators yet to be developed. Because the different universities contributing to AEROWORKS have each made their own manipulators, but may also want to try out new types, the software should be flexible and reconfigurable. Being able to develop a manipulator, without having to spend much time on the software needed to control it, reduces the time it takes to try out a manipulators.

3.2 System overview

The full system, including the drone, has been depicted in Figure 3.1. This is how the system will eventually operate, outside testing environments. The drone carries a manipulator, which can any arbitrary AEROWORKS manipulator, as well as a microcontroller that drives the motors, and it houses a computer (Intel NUC) capable of running ROS. The majority of the ROS nodes have to run soft real-time, and therefore these run on the drone. The UI does not have to perform real-time computations, and can therefore run on a PC connected via a wireless connection. There, an operator could control the manipulator if required.

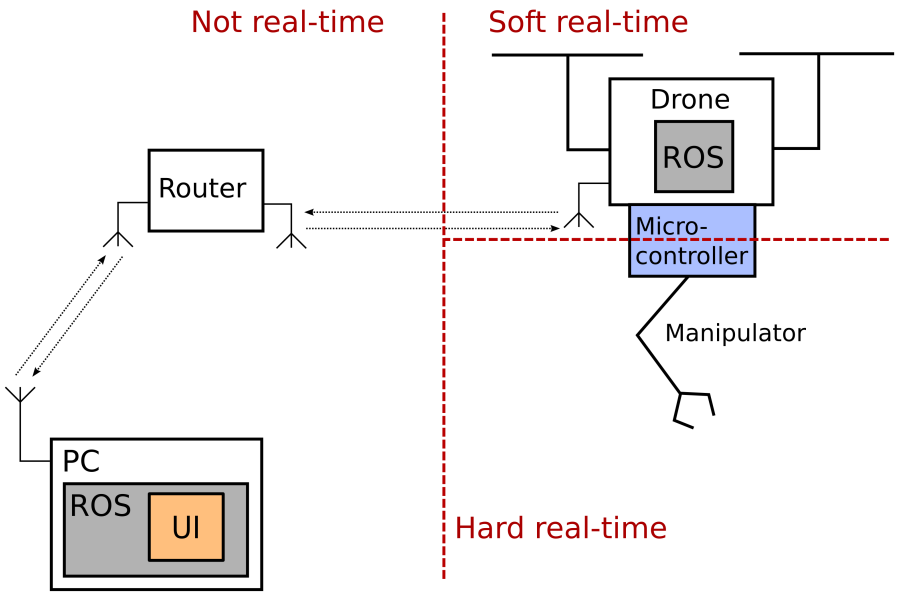

Figure 3.1: The complete system, including the drone, manipulator and PC.

3.3 State at the start of the project

The structure and functionality of the software at the start of the project, that is used as a basis to work with, will now be discussed briefly. It consists of three major nodes, task-level, high-level and low-level control, each of which has a different use in the system. The software structure is shown in Figure 3.2. Services are green (1), nodes are white (2), GUIs are orange (3)

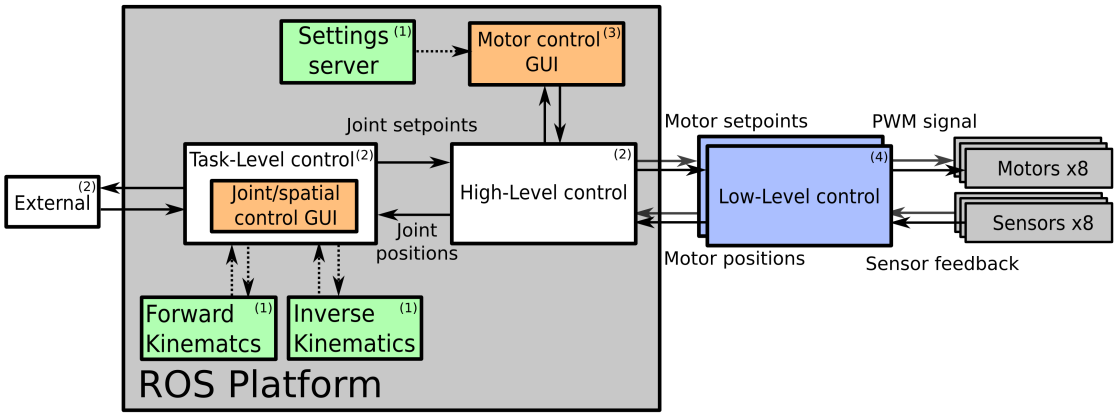

Figure 3.2: Diagram detailing the state of the software at the start of the project.

and the microcontroller-related blocks are blue (4). Arrows represent ROS messages, dashed arrows represent data flow in service calls.

Low-level control houses the actual control-loop and code that drives the motors. It is programmed on a microcontroller, the Arduino Mega 2560, which can be programmed with a simplified version of C++. This microcontroller supports ROSserial, a package that allows serial communication with the rest of the ROS framework. Next to driving the motors, the microcontroller reads sensor data to supply the PID controller with, and also sends this data to the ROS system. The only encoder that is supported is the rotating step encoder, which is an issue if a manipulator uses potentiometers to track the position of its motors.

High-level control deals with the conversion between task-level and low-level control. It is programmed in python, like all other ROS nodes and services. Joint setpoints need to be transformed to motor setpoints using a transformation matrix. This works vice-versa, as motor positions are converted to joint positions in the feedback path. For this reason, transformation matrices come in forward and inverse variants.

High-level will be responsible for handling trajectories in the future. Roughly, it will sequence the setpoints in the trajectory and time them, before converting them to motor setpoints one by one. It is beyond the scope of this project to implement trajectory handling, so this will not be done.

Task-level performs the kinematics calculations that convert Cartesian space to joint space, as was mentioned before. Cartesian setpoints are converted to joint space using inverse kinematics, and vice-versa using forward kinematics. The kinematic calculations happen right before sending the setpoints to high-level. Furthermore, task-level opens a GUI that acts as the main source of input and displays feedback to the user. This GUI also allows a joystick to be selected as an input, in which case the task-level node will start interpreting joystick input and use it as Cartesian setpoints.

To switch between manipulators, a GUI is available where settings can be changed manually, like transformation matrices and PID gains. These custom settings, however, are never stored anywhere and lost after applying them. The result is that each time the software is executed, default values are set. The inverse matrix cannot be updated, as this was not implemented.

The software's purpose is to specifically control manipulators used by AEROWORKS. It can be assumed those are rather simple, lightweight arms, since they will be mounted under a drone. Eight motors are thus assumed to be the maximum amount, controlled by up to two microcontrollers.

3.4 Requirements

The software is required to be universally applicable to at least the manipulators used in the AEROWORKS project, but preferably more. Multiple microcontrollers of different kinds should be compatible with the software. Naturally, different manipulators have different kinematics. These should be reconfigurable in the software, to fit any manipulator.

Some manipulators use mechanics such as differential gears, that allow one motor to influence multiple joints. Vice-versa, joints might be influenced by multiple motors. For this reason, a transformation matrix is required to capture these mechanics and convert between joint and motor space.

High reconfigurability is required. It is important that the software is expandable with possibilities to add manipulators with additional functionality that may be developed in the future. Switching between compatible manipulators should be easy to do, and adding support for additional manipulators should require minimal adaptations and effort.

3.4.1 Core features

Some software features are always required, independent of the manipulator that is to be controlled. These provide core functionality, and include the following:

- Kinematics calculations
- Transformation matrices
- Encoder readings
- Motor drivers
- Control loop
- ROS message system

When switching between manipulators, some of the components named above stay conceptually the same, but their content changes. For example, some form of kinematics is always required, but the actual calculations depend fully on the physical properties of the manipulator. The content should be provided by the user if they want to use a manipulator with certain kinematics. This content includes the actual values and calculations of the above named components.

Whenever any feature is required by at least one manipulator, the software should be able to provide it. Not all manipulators require for example setpoint minima and maxima, but since there are some that do, the software should always be able to apply minima and maxima to joint and motor setpoints. This can easily get out of hand when the variety of manipulators increases, which is why the software will be limited in what manipulators it supports.

A GUI is not in the list, since it is not considered to be a core feature but more like an extension. Still, even though using a GUI is optional, it should function for every manipulator with any number of motors to be fully manipulator independent.

3.4.2 Specific Requirements

Having discussed the purpose of the software and its different components, more specific requirements can be set. They are sorted by priority using the four tags in the Moscow principle, *must*, *should*, *could* and *will not*.

- The software must be able to control a wide variety of different manipulators. Specifically, manipulators that are used in the AEROWORKS project, and might be developed in the future.
- It must be possible to store manipulator specific settings in a config file that is loaded into the software. This will allow far easier switching between manipulators, and settings can be saved this way.
- Low-level control must be able to switch between linear encoders and rotary step encoders, depending on which is used by the manipulator.
- There must be a possibility to run a GUI independently alongside the software, for manual control. The user can then decide whether or not this functionality should be enabled.
- The code must follow a consistent and intuitive naming convention. This software is going to be used by members from different universities, and readable code supports understanding of the software.
- The GUI should be clear and intuitive. No idle buttons or sliders should be present on the GUI.
- It should be possible to connect microcontrollers that do not support ROSserial. Being able to use a microcontroller that does not support ROSserial increases the amount of compatible microcontrollers, and allows manipulators that use those to be controlled.
- Joystick support could be included. It has proven useful in the past to have such functionality, but it is no main necessity.
- The future implementation of trajectory control could be taken into account.
- Kinematic servers for manipulators will not be provided along with the software, only a server template. It is up to the user to fill this in with required kinematics. Because almost every manipulator has different kinematics, covering them is simply too much work.
- Code for low-level electronics other than the Arduino Mega 2560 will not be written. At the moment, low level electronics consist of the Arduino Mega 2560, used in combination with Polulu motors, and therefore the low-level code is written for this scenario.

4 Design and Implementation

In this chapter, several important parts of the restructured software are discussed, regarding the choices made in designing and implementing the software. A diagram of the restructured software can be found in Figure 4.1. Each software component found in the diagram is treated in a separate section.

Again, services are green (1), nodes are white (2), GUIs are orange (3) and microcontroller-related blocks are blue (4). Arrows represent ROS messages, dashed arrows represent data flow in service calls. The encoder subclass is included.

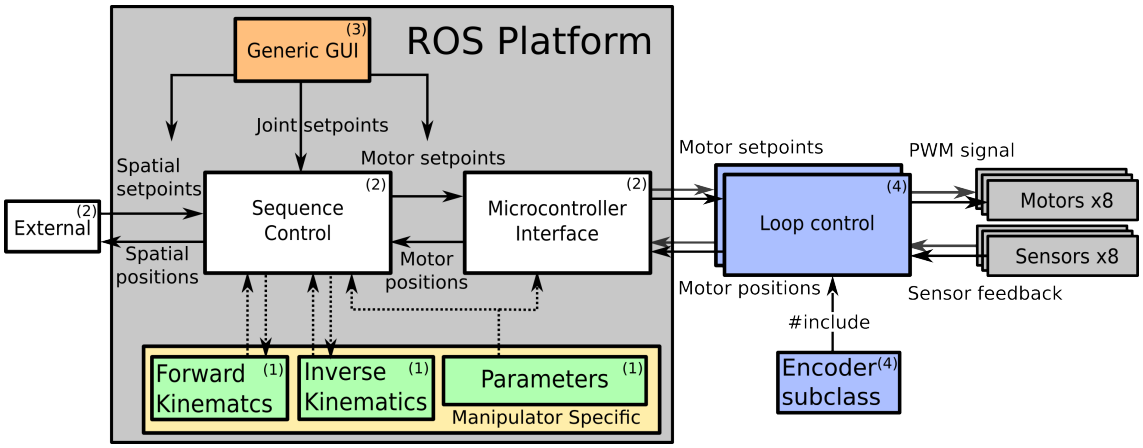

Figure 4.1: Diagram detailing the restructured software.

4.1 General decisions

The manipulators that this software aims to control are drone-mounted manipulators. Some assumptions have been made to save time on implementation, such as the fact that no more than eight motors will be supported, and no actuators other than motors are assumed to be used.

In order to make a software reconfigurable, parts of it should be easily interchangeable, without having to modify other components. By standardizing components, different implementations can be designed for them, which can then be swapped into the software to alter its behavior (Brodskiy, 2013). This will allow the software to be reconfigured and support different types of manipulators. It also allows for a lot of freedom in these implementations. As long as a component meets the operational requirements of its standardized form, it does not matter how it is implemented, and its implementation will not affect the rest of the system.

The general structure of high-level and low-level will be reused, since it provides a clear separation of system tasks, and keeping this separation allows one piece to be considered at a time. Reusing this structure also saves time in designing a new one. The nodes however, were renamed to better reflect their task in the software; low-level control was renamed to loop control, and high-level control was renamed to sequence control. Both will be discussed further in the chapter. Task-level control was left out of the software altogether, as it turns out not to serve any purpose after the software was refactored. It is still possible to run the node along the software, but it is currently idle.

Loop control is programmed on a microcontroller, and thus likely to change, as there are many different microcontrollers available. There is currently at least one other manipulator in the AEROWORKS project that does not use the Arduino Mega 2560 to drive its motors. It is, how-

ever, rather ambitious to take into account all candidate microcontrollers. Minimizing specific microcontroller dependencies in the system has been attempted by introducing an interface between the microcontroller and sequence control, which will be discussed later in this chapter.

4.2 Config file

A config file that contains all manipulator specific parameters was added to the software, written in YAML. This language is recognized by ROS' parameter server, and loaded into the system by the launchfile. This launchfile is also responsible for starting the other nodes and kinematics servers. Once loaded into the parameter server, any node can access the parameters defined in the YAML config file. An additional benefit of YAML is that it is easy to read and understand for humans.

Parameters included in the config file are the following manipulator specific settings:

- Manipulator name
- Number of motors
- PID gains
- The transformation matrix, relating joint space to motor space
- Encoder type
- Encoder factors
- Joint initial positions
- Joint minimum/maximum values
- Motor minimum/maximum values
- Names of the forward and inverse kinematics servers

These parameters will be discussed in this chapter with their respective node. Kinematic server names in the config files are not the actual kinematics, but references to the name of the kinematics server. The actual calculations are not included because they take up a lot of space and would obfuscate the config file. Instead, as mentioned before, these calculations are performed by separate ROS services which are addressed by the sequence control node.

4.3 Sequence control

Sequence control performs conversion from spatial to motor setpoints, and from motor to spatial positions. It calls the kinematic servers that were previously called by task-level. The call was moved to sequence control so that all setpoint conversions happen in the same node. The actual calculations are performed by the service, the kinematics server, and not by the sequence control node. A ROS service behaves just like any function. It takes arguments, and returns a response.

The reason why kinematics are calculated in a separate server, and not in the sequence control node itself, is to improve the reconfigurability of these calculations. As there are many different manipulators, there could eventually be many different kinematic servers, containing the calculations for each of the manipulators. To select the corresponding one, the name of the kinematic server to be called can be specified in the config file. By having them in a separate server, they can be easily replaced by different kinematics while still essentially providing the same functionality. A 'default' kinematics server is provided, but it simply forwards its inputs

to its outputs. Since almost each different manipulator has different kinematics, it was decided to let the user implement these, and the default server serves as a template.

Next, the sequence control node further converts these joint setpoints to motor setpoints, using the transformation matrix specified in the config file. The size of this matrix should always be n by n, where n equals the number of motors. This allows any joint to relate to any motor, and vice-versa. As said before, these conversions happen in both ways, so the inverse matrix is calculated as well, to convert motor positions back to joint positions.

4.4 Microcontroller interface

The microcontroller interface is a connection between sequence control and loop control, and forwards motor setpoints to the correct microcontrollers. It receives up to eight setpoints, one for each motor, and splits those between up to two microcontrollers. It also fetches parameters from the parameter server, and forwards these to their respective microcontroller. This was implemented because not all microcontrollers are capable of fetching those parameters themselves, some microcontrollers do not support ROSserial.

So, even though the Arduino 2560 is capable of fetching parameters via ROSserial, the microcontroller interface reads them from the parameter server, and sends them to the Arduino. Another reason why this is required, is because the Arduinos run exactly the same code, and do not know which of the two Arduino they are. Therefore, they cannot decide for themselves which parameters to import from the parameter server. Difference between one Arduino or the other is made by the interface, using a separate namespace for each Arduino. A benefit of this implementation is that only one Arduino program has to be written, and that Arduinos can be interchanged at will. It also allows the sequence control node to output a single array with setpoints, without taking the number of microcontrollers into account.

4.5 Loop control

Loop control was programmed on a microcontroller, because it houses the control loop which needs to be hard real-time. Messages send in ROS have no guaranteed time of arrival, and are thus not hard real-time. As can be seen in Figure 3.1, part of loop control is soft real-time, and part is hard real-time. The control loop is strictly timed, and any time left is used to listen for setpoints and publish current positions. This ensures the control loop has top priority and does not have to wait for ROS communications.

Even thought the number of motors can vary, the Arduinos always assume there are four motors. If there are more motors, a second Arduino will handle the other motors, and if there are less than four motors to be controlled, the gains of the unused motors will be zero. The decision to let the Arduinos assume there are always four motors is for simplicity. It means not having to check how many motors there are, and deducing how many a particular Arduino has to drive. If for example six motors are used, one Arduino drives four motors, and another drives two. An Arduino can thus not simply use the number of motors defined by the config file. Keeping the number of motors fixed at four does not cause any significant limitations.

Since it was assumed that Polulu motors were used in combination with the Arduino Mega 2560, the loop control software was written for this scenario. In order to drive motors other than Polulu motors, the microcontroller might have to be reprogrammed or replaced, and as a consequence the microcontroller interface as well.

4.6 Physical constraints

Manipulators may have physical constraints, that prevent it from moving beyond a certain point without breaking. These constraints exist in three domains, Cartesian space, joint space and motor space. They will each be treated separately.

Motor constraints are applied to the setpoints received by the microcontroller. They prevent the motors from moving beyond a physical constraint to prevent damage to the manipulator. Potentiometers, for example, can only move a certain distance before reaching a physical stop. The AEROWORKS project does not have the budget to afford expensive failures, so they must thus be prevented. This safety layer is thus applied on the microcontroller, as close to the control loop as possible, to ensure it cannot get bypassed.

Joint constraints are applied in the sequence control node, after receiving joint setpoints from either rostopic or the out put of the inverse kinematics server. Joint constraints differ from motor constraints because some joints may not be able to rotate infinitely whereas the motors controlling them theoretically can.

Constrains in Cartesian space include any objects in the environment, including the manipulator itself, that the manipulator should not move through. A combination of joint angles that pushes the end-effector into a drone's may not occur, and thus some constraints can be set. These, however, are not taken into account by the software explicitly, however they can be build into the kinematics server. When performing inverse kinematics using a certain position in space, a check is performed to see whether this position is reachable at all. Additional checks can be build in to see if the manipulator is not moving through itself or the drone.

4.7 Switching encoders

To track motor positions, the microcontroller uses encoders. The two types supported by the software are linear encoders and rotary step encoders. Not every manipulator uses the same encoder, and therefore these need to be reconfigurable.

The encoder code has been implemented using a superclass and one subclass per encoder as can be seen in Figure 4.2. The superclass contains virtual functions that each encoder uses, such as 'getValue' to read the current value. The actual implementation of 'getValue' is done by the subclasses, with every subclass having their own implementation of these functions. Setting the encoder type in the config file determines which encoder subclass is loaded by the Arduino. The Arduino code can then call the superclass, and will address the functions of the chosen subclass.

By instantiating both classes, but addressing only one of them, the class name 'encoder' can be used throughout the entire code. The name is independent of the class actually selected and class selection can happen during runtime. It circumvents many switch cases and if-statements to choose the function to be executed by re-using function names as well. For example, superclass function 'encoder->getValue' will always return the angle of the motor, using the function defined in the selected subclass. Since a superclass can have multiple subclass implementations, if additional encoders are required, these can be implemented as a new subclass and included in the microcontroller code.

The reason why this implementation was chosen is because different classes cannot share the same name, and conditionally initiating a certain class is not straightforward in C, as creating an object within an if-statement generally does not create it permanently. A previously used solution was to comment-out certain include statements to prevent code from executing, but this is an unusual way to achieve reconfigurability, and requires the code to be re-uploaded whenever switching encoder classes.

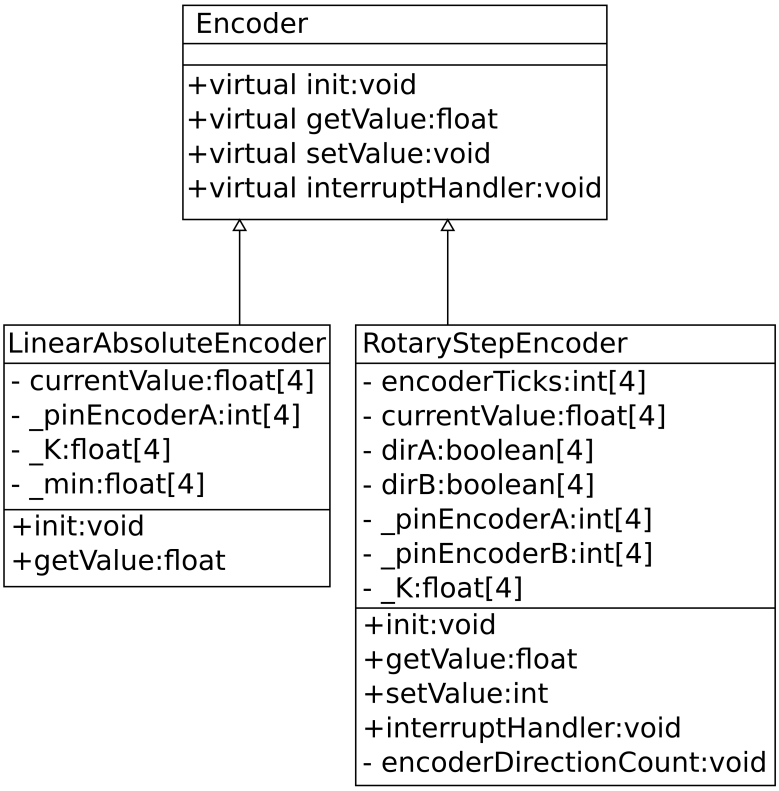

Figure 4.2: The encoder-specific subclasses as implementations of a generic encoder superclass

4.8 External

With 'external', anything that connects to the software for either input or output purposes is meant. The GUI is technically also part of external, and was included as default I/O component so that the manipulator can be intuitively controlled by a human operator. It will be discussed in this section as well. It was explicitly shown in Figure 4.1 to show where it sends its ROS messages.

Joystick control could also be an external extension, which would allow a joystick to be used to move the manipulator. If it were to be implemented, it would be best to add it as a separate node such that it can be disabled when not required, and easily replaced to support different types of joysticks. Since it is not part of the software, and since there is currently no need for joystick control, this node was not implemented.

End-effectors can be considered part of the manipulator, however, there is too much variation between different end-effectors to consider all of them. Therefore none have been taken into account, and to control them a separate, end-effector specific node should be written. This is thus not considered to be part of this software, but 'external'.

Rostopic is also considered to be external, since it allows users to publish messages to topics directly via the terminal. However, it does not only connect to sequence control as might be suggested in Figure 4.1, simply because any node in the system can be addressed by rostopic. In general, any node can publish to any topic if programmed to do so, and thus reach any node subscribed to said topic. This means that a joystick node, mentioned before, could connect to sequence control, but also to the microcontroller interface, depending on how it is implemented.

4.8.1 A generic GUI

For simple control purposes, a generic GUI was made that can send messages over ROS to the motor, joint and spatial setpoint topics, and receives position information from the three domains. This allows the user to control the manipulator from three domains, as well as use the current manipulator position as setpoints.

The GUI scales to support up to eight different motors/joints and unusable sliders are hidden from the user. Input in three domains is accepted and the GUI allows to switch between those. In this way the manipulator can be controlled from one domain at a time, preventing conflicting inputs. It also avoids having three GUI windows open at once, which would take three times as much space on the screen.

Glade was used to build the GUI (project, 2016), which was then connected to the ROS framework via the GUI node. Glade was used because it allows quick creation of professional interfaces, without having to spent much time on coding. This saves time on both learning how to implement the GUI and the actual programming.

5 Results

Core functionality and manipulator specific settings were successfully separated, allowing the latter to be edited without having to deal with the first. In this chapter a close look will be taken at the finished software as a whole. It will be compared to the requirements that each manipulator presents it with, to see if it is sufficiently universal.

5.1 Application to two manipulators

AEROWORKS currently uses two manipulators that utilize Polulu motors and Arduino microcontrollers. These are the UT manipulator, and CARMA by the LTU. In evaluating the final software, they will be considered first. Due to complications, the actual manipulators could not be used for testing, and only the individual motors were used to see if the software functioned properly. It was still possible, though, to test the software for reconfigurability without them. To test motor movement, the Polulu motors were connected to the microcontroller without manipulator. The following paragraphs will prove whether the software is indeed sufficiently reconfigurable, and capable of controlling the manipulators used by AEROWORKS.

The UT manipulator uses up to five motors to control X, Y, Z, yaw and pitch, where yaw and Y are optional, and CARMA uses only four motors. Since the software can accept up to eight motors and eight joints, all of these degrees of freedom can be controlled.

The fact that the two manipulators are different physical structures poses no problem, because the software can be configured using any kinematics server. Even the fact that one manipulator operates in two spatial dimensions and another in three, does not influence the software's ability to properly control the manipulators, since the kinematic servers can have up to eight input and output arguments. As long as the kinematics servers are written correctly, the location in space is used to calculate the required joint movements, and vice-versa. Thus, no matter how complex the movement of the manipulators is, as long as the proper kinematics servers are included, this conversion can be performed.

Multiplying these joint setpoints with the transformation matrix results in motor setpoints. This matrix can be of any size, although other parts of the software currently limit this to eight by eight. This allows all linear joint-motor relations to be represented. The UT manipulator uses two differential gears, whereas the LTU manipulator uses belts that cause all but one of the motors to control two joints. All of these relations can be entered as a matrix in the config file, so as long as motor and joint space are linearly related the software supports it.

Both the LTU and the UT manipulators have physical constraints on their joints. These depend on physical properties of the manipulator and may differ between manipulators. Physical motor constraints, however, are only present in the LTU manipulator, limited by the potentiometers it uses to track its motors. The rotary encoders used by the UT manipulator allow infinite motor rotation. The constraints are supported by the software through the config file and ROS parameter server in both joint and motor space.

Potentiometers and rotary encoders are supported. These are independent of the manipulator and can be chosen and altered using the config file. The ratio between the encoder and the actual motor angle can be set there, and when an encoder requires the initial motor positions to function, these can be supplied as well. This is the case with the rotary step encoders, which can only detect a change from their initial position.

Thus, the software is sufficiently reconfigurable to support both of these manipulators, using the Arduino Mega 2560 microcontroller. Reconfigurations that need to be made in the event a different microcontroller is used, are discussed in the following paragraphs. Again, in the

software, different microcontrollers have not been taken into account, but it is still possible to make some microcontrollers compatible.

5.2 A different microcontroller

The manipulator designed by ETH is slightly different in that it uses different motors and a different microcontroller to drive them. The encoder classes written for the Arduino Mega 2560 can not be used by this microcontroller, and it is not capable of using ROSserial. It is still possible to use it with the software though, by reconfiguring the microcontroller interface. It should be changed in two ways.

First, the microcontroller interface should be adapted to utilize a communication protocol accepted by the microcontroller, in order to be able to send messages to it. It is the user's responsibility to ensure that the microcontroller handles these messages correctly. Vice-versa, the microcontroller should send motor positions back to the microcontroller interface using the same protocol.

Secondly, the distribution of motor setpoints might have to be restructured. The ETH manipulator uses only three motors, and thus it can be assumed that one microcontroller is capable of driving those. It is then only necessary to send three setpoints. The encoder-specific code on the microcontroller can be made reconfigurable in the same manner as was done for the Arduino Mega, but this is entirely up to the user.

So initially the software is not compatible with microcontrollers other than the arduino, but it can be made so by replacing the microcontroller interface, which is partly the reason why it was implemented in the first place. This allows the user to use their own microcontroller without having to touch the sequence control node.

6 Conclusion and future work

All in all, to support an entirely new manipulator, several reconfigurations are required. The more alike it is to an already supported manipulator, the fewer changes need to be made. The components that are different for almost all manipulators are related to its physical properties. These include the kinematics server, transformation matrix values, motor and joint minima and maxima and motor gains. Independent of the manipulator, the type of encoder should be specified, as well as the factors between sensor output and actual angles. If the encoder is not supported yet, a subclass should be written for it, and included in the microcontroller code. Finally, a launchfile should specify what to run and what not, when booting the software. Here, the config file is loaded and the kinematics servers are started. An elaborate explanation on how to reconfigure the software can be found in appendix A.

It was shown that with some effort the software is capable of controlling all of the AEROWORKS manipulators, and that new kinds of manipulators can be supported by it, due to its high reconfigurability. Most microcontrollers can be made compatible with the software through some modifications to the microcontroller interface, which need to be made only once per microcontroller when done properly.

6.1 Software comparison

Now, the state of the software before and after the project will be compared, and the improvements made will be summarized. These improvements were made to the structure of the software and the code itself. Major improvents were the following:

- A config file has been added to configure certain parameters and save configurations.
- In the previous software, the transformation matrices had to be four by four, whereas now they can be of any size. This allows more complex motor-joint relations to be represented by it. The software now also calculates the inverse matrix.
- Different kinematics servers can be called just like the previous software could, but now the number of arguments that are passed to and received from these servers is variable. Thus, instead of always having to call kinematics with three DOFs, a kinematics server can be called with for example six DOFs.
- A microcontroller interface was added to accommodate generic motor setpoints for certain microcontrollers.
- Encoder classes were implemented to allow switching between encoders without having to re-upload code to the arduino, as well as easily adding different implementations of the encoders.
- Minimum and maximum value constraints in both joint space and motor space were not present in the previous software and were added for safety reasons.
- The GUI has reduced three to four times in size, depending on the amount of motors used, and runs in a separate node so that it is not mandatory to run it with the software.
- The overall code was clarified with commentary, and many unused lines of code were removed.
- Names of objects were unified to follow the same naming convention wherever possible. Certain functions for example cannot be renamed.

No changes were made to the way the motors are driven, or to the code that drives the motors, and thus there is not much difference to be seen when looking purely at the motors. In general, the code is more readable than it was at the start of the project, and this should make it easier for future programmers to understand it and perhaps implement additional features, some of which will be considered in the next section.

6.2 Future work

The GUI is currently build in Glade, saved as a Glade file, and addressed by a python script. Building the GUI could also happen within this python script, in which case no Glade file is required at all. Instead of using a fixed build provided by the Glade file, a variable build can be made. This allows any number of sliders to be created, but more importantly, it removes the need of having the corresponding Glade file available to run the GUI.

One of the ROS features, rqt reconfigure, can be used to dynamically alter node parameters without having to restart the node. This was not implemented in the software since it is currently not required to alter parameters during runtime, but this functionality might prove useful in the future. For example, if the manipulator should perform slower, more accurate movement, the gains of the motors could be adjusted during runtime to achieve this.

A feature missing from the current software is trajectory handling in the sequence control node, allowing trajectories to be received and sequenced as separate timed setpoints. Other features that are not present are velocity and acceleration control. It might be required for some tasks to move the manipulator at a constant speed, which would make velocity control useful.

Currently, joint constraints are not applied when sending setpoints directly to the microcontroller or microcontroller interface node. The constraints are applied to joint setpoints, in the sequence control node, before transforming them to motor setpoints. After that, they are no longer taken into account. To be completely safe, the microcontroller should convert the motor setpoints back to joint space, apply joint constraints, and then convert them back to motor space to apply motor constraints. This rather tedious task has not been implemented on the microcontroller, and thus care has to be taken when controlling a manipulator from motor space.

The microcontroller interface could be made to support more communication protocols than just ROSserial. This would increase the range of compatible microcontrollers. To prevent it from becoming a too large program, it may be best if multiple, separate interface nodes are made, one for each required protocol or microcontroller, instead of one single node that supports them all. The launchfile can then be coded to only run the required interface.

A Adding a manipulator

The following text contains documentation on how to setup a manipulator (existing or new) with the software, so that it can be controlled. It requires a variable number of steps, depending on how much the manipulator differs from what is already supported. This will become clear in the next paragraphs.

A.1 Reconfiguring files and servers

The different components that always have to be configured in order to run a manipulator are:

- The launchfile
- The config file
- Kinematic servers

More radical changes can be made to other components, but these require modifying some code, and do not always require reconfiguration:

- Encoder classes
- Low-level interface
- Microcontroller code

All of these components will now be treated one by one.

A.1.1 The launchfile

To start the software, the launchfile is executed, which in turn starts up all the different nodes and servers. It also loads the contents of the config file into the ROS parameter server. This launchfile has to be made specifically for the manipulator that is to be controlled. A few things should be taken care of:

- Which of the config files is loaded into rosparam, is defined here. This allows multiple config files to be simultaneously present.
- The system nodes such as high-level and the low-level interface should be started, as well as the required kinematics servers, depending on the manipulator used. There might be multiple kinematics servers present, if multiple manipulators have already been implemented.
- The microcontrollers should be connected to the ports specified in the launchfile, or these ports should be changed.
- The microcontroller namespaces can be set here, which should correspond to the topic name the low-level interface node publishes to. This allows two arduinos running the same code to work communicate on separate topics.

A.1.2 The config file

The config file contains parameters that are loaded into rosparam at startup, to be accessed by the different nodes in the system. These values can be edited in the YAML file which will be loaded the next time the software is started. These are straightforward but important values, such as the contents of the transformation matrices. The type of encoder used is selected here

too, as well as the name of the kinematics server used. Note that this is not the actual kinematic server name, but the name of the .svr server file in the servers folder of the package. An example of a config file is shown below.

```
manipulatorName: UT Manipulator
numMotors: 4
gains:
    PGains: [150,150,150,150]
    IGains: [120,120,120,120]
    DGains: [80,80,80,80]
transformationMatrix:
    - [1,0,0,0]
    - [0,1,0,0]
    - [0,0,1,0]
    - [0,0,0,1]
encoderType: 1
encoderKFactors: [0.00698,0.00698,0.00698,0.00698]

jointInitials: [2,2,2,2]

jointMin: [0.0,0.0,0.0,0.0]
jointMax: [0.0,0.0,0.0,0.0]
motorMin: [-5.0,-5.0,-5.0,-5.0]
motorMax: [5.0,5.0,5.0,5.0]

forwardKinematicsSrv: FwdKinCalculation
inverseKinematicsSrv: InvKinCalculation
```

Some care should be taken not to insert conflicting data, such as setting the number of motors to 4, and entering a 3 by 3 matrix. The latter would imply only 3 motors are used, and the software is not able to handle this. Also note that most of the values are arrays. These should have exactly as many indices as there are motors because each index represents a motor. The transformation matrices is actually a nested list, but representing it in this way makes it more clear to the user. Additional rows and columns can be added if required.

A.1.3 Kinematic servers

The kinematic servers are the manipulator specific kinematic calculators, that transform from spatial to joint setpoints and from joint to spatial positions. They should do exactly that, accepting an input array and outputting an array as well. Taking the inverse kinematics as an example, it accepts an input array containing spatial setpoints as argument (X,Y and theta, for example) and uses these values to calculate the corresponding joint positions, which it outputs as an array. Note that the order and amount of input and output values in the arrays should be consistent with the rest of the software. This is the responsibility of the user.

A.1.4 Encoder Classes

When switching to a different encoder, the one that user wants to use can be selected in the config file. This is less trivial when the encoder to be used is not supported by the software, in which case it has to be implemented first. This requires a few steps to be taken.

First, the encoder subclass should be written, consisting of a .h and a .cpp file. This class is a representation of the encoder parent class, and can only use the virtual function defined by the parent superclass as public functions. These functions can be given a custom implementation

to suit the user's needs, and will be executed by the low level code to achieve tasks such as reading the encoder value.

Secondly, the encoder subclass must be instantiated in the low-level software. Each subclass should always be instantiated, independent of whether it is actually used or not. The subclass that is ultimately used is defined by the switch case, where every encoder type has its entry. An entry should be added for every new encoder subclass. The subclass is then instantiated inside this case, which is executed based on the selection in the config file. The encoder superclass is populated by the functions defined by the selected subclass, and all works as expected.

Finally, the code should be re-uploaded to the microcontroller, which has to be done only once for every new addition.

A.1.5 Low-level interface

When using a manipulator that uses a different distribution of motors, or should deliver set-points to a different kind of microcontroller requiring a different communication protocol, the low-level interface should be modified. This is not supported by the software, and thus requires arbitrary modifications depending on the user's needs. In the end, the low-level interface should accept the standard ROS messages that high level sends, and distribute these over the used microcontrollers. The situation at hand determines the implementation.

Important is that all available functionality is properly forwarded. Motor maxima and minima, encoder factors, PID gains and the encoder type are all supplied by this node regardless of the microcontroller used, and this

A.1.6 Microcontroller

Using a different microcontroller is possible, but requires it to be entirely programmed by the user. Also the low-level interface should properly connect to it, as explained in the previous subsection. Most likely, the encoder classes used by the arduino cannot be used on other microcontrollers and has to be reimplemented as well. Other microcontrollers can thus be made compatible if really necessary, but are not initially supported.

Bibliography

AEROWORKS (2016a), "AEROWORKS 1st Integration Week Primary Results".
http://www.aeroworks2020.eu/videos/

AEROWORKS (2016b), "Collaborative Aerial Workers".
http://www.aeroworks2020.eu/

Brodskiy, Y. (2013), *Robust autonomy for interactive robots*, ISBN 978-90-365-3620-2, doi:10.3990./1.9789036536202, p. 26.
https://www.ram.ewi.utwente.nl/aigaion/attachments/single/1174

Dekker, G. (2015), "Mechanical design, prototyping and evaluating of a 3 DOF manipulator for an Unmanned Aerial Vehicle", pp. 18–26.
https://www.ram.ewi.utwente.nl/aigaion/attachments/single/1304

Kominiak, D. (2016), "CARMA - Compact AeRial MAnipulator".
http://www.ltu.se/staff/d/darkom-1.92329?l=en

project, T. G. (2016), "What is Glade".
https://glade.gnome.org/index.html/

ROS.org (2016), "Core Components".
http://www.ros.org/core-components/

Schipper, G. (2016), "Mechatronic Design of an Aerial Manipulator", pp. 11–18.

Wuthier, D. (2016), "Dual Controller Design of a Manipulator Arm Mounted on a Multirotor".
http://stisrv13.epfl.ch/masters/img/557.pdf