A distributed navigation and guidance system for autonomous vessel

NIKLAS ROLLEBERG
A distributed navigation and guidance system for autonomous vessel

NIKLAS ROLLEBERG
Abstract

Using distributed micro-controllers in an autonomous vessel can result in higher robustness and lower energy consumption compared to a traditional centralized approach. However adapting traditional software (e.g. navigation and guidance systems) for a distributed micro-controller system is a complex and challenging task.

In this thesis a software solution for controlling autonomous vessels with on board distributed micro-controllers is presented. It is capable of controlling the vehicle by itself, including routing around obstacles in moderately complex environments, or act as an interface to another computer. The routing capabilities come from the use of a high-level path-planner based on RRT, and a low-level vehicle controller based on potential fields to follow the path from the path-planner. By using these methods together the software is capable maneuvering a sailboat between waypoints.

In our experiments distributed computing is investigated for the path planning and evaluated in terms of computational time for 1,2 and 3 nodes. A parallelization technique called OR parallelization was tested and it offered a reduction in computational time by 27% with two nodes and 36% for three nodes against a single node. Nevertheless, this gain may not be, significant enough to warrant the extra complexity.
Sammanfattning

Att använda distribuerade mikro-controllers i en autonom farkost kan resultera i ett robustare och energisnålare system jämfört med ett traditionellt centraliserat system, dock så är det en utmanande och komplex uppgift att anpassa existerande mjukvara för navigation och styrning till ett sådant system.

I denna rapport presenteras en mjukvarulösning för styrning av autonoma farkoster med ett distribuerat system av mikro-controllers som finns ombord. Mjukvaran kan antingen kontrollera farkosten själv (inklusive ruttplanering runt hinder i måttligt komplexa miljöer), eller fungera som gränssnitt till ett annat system. Ruttplaneringen görs av en planeringsalgoritm som heter RRT tillsammans en lågnivåkontroll, som använder sig av potentialfält för att följa rutten från planeraren. Med dessa två steg så kan mjukvaran styra en segelbåt mellan givna platser.

Distribuerade beräkningar undersöks för ruttplaneringen och utvärderas i termer av beräkningstid för 1,2 och 3 noder. En parallelliserings- teknik som heter OR-parallellisering testades och gav prestandavinst på 27% respektive 36% för 2 och 3 noder. Författaren tycker dock inte att den prestandavinsten är hög nog för att väga upp den extra komplexitet som den medför.
# Contents

1 Introduction 1
   1.1 Research Question 2
       1.1.1 Specified problem description 3
   1.2 Objectives 4
   1.3 Research Methodology 4
   1.4 Outline 5

2 Background 6
   2.1 Micro controller interface 6
   2.2 Vehicle control software architecture 7
   2.3 Command filtering and error detection 7
   2.4 Localization 8
   2.5 Planning 8
   2.6 Control 9

3 Related work 10
   3.1 Software architecture 10
   3.2 Planning 11
       3.2.1 Dijkstras algorithm and A* 11
       3.2.2 RRT* 12
       3.2.3 Distributed RRT 14
       3.2.4 Potential field methods 16

4 Method 19
   4.1 Software architecture 19
   4.2 Logging 20
   4.3 Communication with sensors and actuators 20
   4.4 Sensors 21
   4.5 Localization 22
   4.6 Missions 22
4.7 Planning ........................................... 22
  4.7.1 Maps of the free-space ......................... 23
4.8 Distributed planning .............................. 24
4.9 Command filtering ............................... 24

5 Implementation .................................. 25
  5.1 The captain - network of TEENSY and ARDUINO ... 25
    5.1.1 Communication .................................. 25
  5.2 Surface vehicle - MARIBOT VANE ................. 26
    5.2.1 Differences from a traditional sailboat ........ 26
    5.2.2 Sensors .......................................... 26
    5.2.3 Low level path planning ......................... 27
  5.3 Underwater vehicle - MARIBOT LOLO ............... 28
    5.3.1 Sensors and actuators .......................... 28

6 Experiments ...................................... 30
  6.1 Simulated tests .................................. 30
  6.2 Sea trials ....................................... 33

7 Results ......................................... 34
  7.1 Simulated tests .................................. 34
  7.2 Sea trials ....................................... 39
    7.2.1 MARIBOT VANE .................................. 39
    7.2.2 MARIBOT LOLO .................................. 41

8 Discussion ...................................... 42
  8.1 Autonomous sailing ............................. 43
  8.2 Path planning ................................... 43
  8.3 Performance gain in a distributed system .......... 44
  8.4 Validity of the Results ........................... 45
  8.5 Ethical and social aspects ....................... 46
  8.6 Future work ..................................... 46

Bibliography ...................................... 47
Chapter 1

Introduction

Autonomous vessels can travel relentlessly for an extended period of time. This makes them ideal for research and data collection. To achieve a long operational time with an autonomous vessel, one of the most important factors is power consumption. Other than an efficient hull it also means that the on-board electronics should consume as little energy as possible. Two autonomous vehicles are considered in this thesis (Figure 1.1), both with strict requirements on power consumption and with similar hardware configurations. The hardware used is distributed micro-controllers for controlling the vehicle and a more powerful research computer for handling research missions. The distributed micro controllers work as an interface to the vehicle and allow the research computer to stay more general, and also to be powered down to save power when it isn’t being used.

![Figure 1.1: vehicles considered in this thesis](image-url)

(a) Maribot Vane  
(b) Maribot Lolo
This thesis presents a software solution for the micro-controllers responsible for controlling the vehicle. They will act on instructions given by either a human or the research computer. The micro-controllers are connected via a serial bus with one master node responsible for receiving and processing the instructions, and several slaves connected to sensors and actuators. All instructions will be filtered to prevent actions that could damage the vehicle. In case something goes wrong, the software will be capable of moving the vehicle to a preset location to be recovered without the need for any instructions.

Achieving this level functionality on a system with as little resources as this is a challenge. A way of determining if the actions of an instruction might put the vehicle at risk is needed, and if it could do so this instruction should not be carried out. There is also a need for being able to detect abnormal behavior of the vehicle and take some appropriate action based on that. For example, if a sensor were to fail, a good action might be to abort the current mission and go to a preset location for recovery. In order to be able to travel to the recovery location, the vehicle must be able to navigate around obstacles, which is challenging when the amount of resources available are severally limited.

Since the computational resources of one micro-controller are very limited, distributing tasks among several of them is a good way increasing the total capability of the system. An added benefit of adding additional intelligence to all nodes in the system is that if the master were to fail the rest of the system could take action and go to the surface to prevent the vehicle from being lost. It would also be possible to have a dormant master node that takes over if more advanced coordination between actuators are needed.

1.1 Research Question

- Can distributed micro-controllers be used to control an autonomous vessel? And is it possible to solve complex tasks like path planning on that hardware?

- Is there any performance to gain in terms of computational time and reliability from using more than one micro-controller in the
1.1.1 Specified problem description

The software is going to be used on different vehicles, so it has to be easy to adapt and still be able to perform all required tasks. In order to improve performance and be able to solve more complex tasks, distributed computing will be investigated by using the micro-controllers connected to the sensors as nodes in a distributed system.

Most of the required tasks like gathering data from sensors and receiving instructions can be solved fairly easily, but there are some that will require much more work. The easier tasks will not be a big part of this thesis since the research value is limited. However, difficult tasks like finding a path from one location to another while taking the vehicle's constraints into account and avoiding obstacles, will be handled more in depth. This is a difficult task to solve on a micro-controller since it usually requires a lot of RAM memory.

The distributed system consists of several micro-controllers. The maximum number of nodes connected to one of the vehicles during this thesis is 10, but the system should be able to handle more if needed in the future. The micro-controllers will be connected to a serial bus with a common master. Multiple nodes will share the communication wires, so it will not be possible to receive data from more than one node at a time. The protocol used will be a request-response protocol with the master controlling when a node is allowed to send data.

Limitations

Although identifying problems with the vehicle that are less obvious than failed sensors is needed, it will not be covered in this thesis. In order to identify such problems good knowledge about the vehicles' normal behavior is needed, which is not available at the time of this thesis.
1.2 Objectives

The objective of this project is to develop software capable of controlling an autonomous boat. It should be general enough to be easy to adapt to new vessels and not too complex to understand how it works.

The software has to be able to execute instructions given by a user or an external computer, either located in the vessel or remotely. If an instruction could put the vessel at risk it should not be carried out. The software should also be capable of returning to a predetermined location for recovery at the end of a mission or in case a mission is canceled due to problems with the vehicle.

1.3 Research Methodology

Firstly the goals with the project are identified and research questions are formulated and defined for the thesis. Then a literature review is conducted to get a view of the field and what solutions currently exist. A solution approach is then selected and modified to suit the project at hand. The feasibility of implementing the approach on the distributed micro-controllers is then evaluated, if it is deemed feasible it is implemented, otherwise the approach is revised or another approach is selected. After implementation brief tests are conducted to check whether the approach could fulfill the goals, if it does, more tests are conducted to verify this, otherwise the approach is altered once again.
1.4 Outline

This thesis is structured as follows: Chapter 2 contains background information about the problem and short introductions to various areas that will be covered later in the thesis. Chapter 3 contains related work. Chapter 4 explains the method chosen, and Chapter 5 has details about the implementation and the vehicles used for testing. Chapter 6 details the experiments conducted and chapter 7 contains the results. The results are then discussed in chapter 8.
Chapter 2

Background

Autonomous vehicles consist of two major components, the vehicle itself and the software system controlling it. An interface between these two components is needed in order to transfer data from the vehicle sensors to the software and send back commands for the actuators. Traditionally all this is either done in the same software program or with a micro-controller program that acts as the interface.

2.1 Micro controller interface

Having a programmable interface between the high-level software and the low-level sensors and actuators makes it easier to use high-level software that is less hardware specific to control the vehicle.

The programmable interface also allows for an extra level of safety since it can prohibit commands that could damage the vehicle from being executed. The more advanced the interface gets, the more safety features can be added. If the interface is capable of controlling the vehicle without any input from the high-level software it allows for return to home maneuvers in case something has failed or if the power level gets critically low, however this also means that the interface will need the same functionality as a rather simple high-level controller would, e.g. Localization awareness, environment awareness and ability to plan and follow a path. In other words the interface would need to be a vehicle controller in itself.
2.2 Vehicle control software architecture

A common way of designing software architecture for autonomous vehicles is by using stand-alone blocks. An architecture like this makes maintenance and testing easy since all parts of the software can be tested separately. One block can be responsible for the localization while another does the control of the vehicle. The localization block will share information about the vehicle’s location with the control block so it can e.g use closed loop-control.

The Robotics Operating System (ROS) is built around this principle with stand alone-blocks. In ROS each block is called a node and is run as a separate program. Information between the nodes is shared via network messages. However ROS is not available nor suitable for something with as lite processing power as a micro-controller. So if a block architecture like this were to be used it has to be implemented in another way.

For less powerful systems like the one being considered in this thesis, functionalities like path-planning are rare. These kinds of systems with micro controllers are commonly used in applications with few features, but high requirements on quick response times. e.g the control of a drone. The application considered in this thesis is the complete opposite. While fairly quick response times would be desirable it is not one of the key requirements since the autonomous vehicles considered are rather large and slow to react to any input. Still, the response times should be fast enough to ensure that the vehicle is controllable in any situation.

2.3 Command filtering and error detection

It is important to be able to detect if an external command could risk damaging the vehicle so it can be prevented. This could, for example be a command which would bring the vehicle close to an island, or in the case with an underwater vehicle a command that would bring the vehicle too deep.

The decision to carry out a command or not should also be influ-
enced by the state of the vehicle. For example if a depth sensor is broken, commands that includes diving should probably not be executed, while surface operations could still be allowed. However if a more crucial sensor like the compass is broken, no external commands at all should be allowed at all in order to allow the operator to recover the vehicle.

2.4 Localization

Knowing where you are is easy if you have access to an absolute positioning system like GPS. GPS positions can sometimes be noisy but can be filtered with a complementary filter if needed. If you don’t have access to any such system you instead have to estimate your position from other inputs. That is the case when traveling underwater due to the rapid attenuation of radio waves in water. One way of doing this is to use a known starting location and do dead reckoning by integrating the vehicle speed and course over ground to estimate the position. Another way is to use Simultaneous Localization and Mapping (SLAM). SLAM was developed for robotics on land as a way of for a robot to create maps of the environment and localize itself within that map at the same time. In SLAM, features are extracted from the environments and the movements of the robot is calculated from the change in relative position of these features. This idea has also been applied to underwater robotics [9]. Another feature based localization technique is to use currents as land marks for localization [5]. This approach is inspired by how fishes localize themselves in the surrounding environment.

2.5 Planning

In order to travel from one location to another where it isn’t possible to draw a straight line between the locations due to obstacles, some kind planning is needed in order to find a traversable path. Planning can be done in different ways. Motion planning is one type of planning. It plans a continuous motion from start to finish while taking the vehicle constraints into account. This is commonly used in robotics where the vehicle constraints are well known and restrictive, e.g a car [4]. A simpler case is path planning, the goal of path planning is just to find a
path from start to the goal, without taking the vehicle constraints into
account, as a result of this it is much less computationally heavy, but
it requires a more advanced control step to follow the path, since it
doesn’t contain any information about how the vehicle and actuators
should move. Since no vehicle constraints are needed, the planning
software doesn’t have to be vehicle specific and can be used on any
vehicle.

2.6 Control

Once a path has been planned, it is up to the control part of the soft-
ware to follow it. If the path is smooth and the vehicle constraints was
taken into account during the planning phase, the path can be used
as set-points for a regulator. However if that is not the case, more ad-
vanced methods might be needed. An example of this is if the vehicle
is a sailboat and the wind direction was not taken into account during
the planning, depending on how the wind direction is related to the
path the vehicle might not be able to follow the path exactly as it was
planned.
Chapter 3

Related work

3.1 Software architecture

The structure of the software responsible for controlling an autonomous vehicle is important. It has to be well thought trough all the way from the way data is acquired from sensors to the way it handles commands from users.

In the paper *Software Architecture for Autonomous Vehicles* [6] the author showcases the software structure of an autonomous catamaran all the way from low to high-level. The low-level software is responsible for gathering data from sensors and controlling the vehicle, much like the software being considered in this thesis. The software presented runs on a Linux computer, which is capable of multi-threading, however the author has chosen to just use two threads; one for controlling the vehicle and gathering sensor data and another for logging. The reasoning behind this is that more threads does not necessarily result in any performance increase, but they do increase complexity and thus the risk of bugs. The control thread has an infinite loop that collects data from a sensor or the user interface, processes it, runs control loops and changes the actuators if needed. This simple approach allows the developers to get a good overview of the software when developing it, and since everything is done in a specific sequence debugging is simplified.

Another way of designing the software program is in blocks/modules, this is suitable for vehicles where complex functionalities and behavior
is needed. In the paper *Towards a Functional System Architecture for Automated Vehicles* [14] the authors discuss several different implementations of this type and their information flow. Implementations with this architecture typically have a hierarchical topography with low-level modules containing controllers at the bottom and higher level modules like planning and mission handling at a higher level. Information typically flows from the modules higher up in the hierarchy to the ones below. This simplifies development of new complex behaviors since operations contained in modules lower in the hierarchy can be reused easily.

### 3.2 Planning

There are endless possible routes between two points in space. If the space has no obstacles and a convex boundary a straight line could be drawn between the points, however this rarely the case. In order to find a path in a more complex environment, other methods are needed.

#### 3.2.1 Dijkstra's algorithm and A*

The shortest path algorithm developed by Edsger W. Dijkstra in the 1950s solves the problem of finding the shortest path between two nodes in a graph. The algorithm finds the shortest path by exploring the graph from the start node in all directions, for each visited node the distance to the start node is recorded, so when the goal node is reached the distance is known.

A* is another shortest path algorithm. It works similarly to Dijkstra’s algorithm except it uses heuristics to speed up the search. The heuristic guides the exploration towards the goal, and thus reducing the number of nodes that needs to be explored. A common heuristic to use is the Euclidean distance between the node considered and the goal.

Dijkstra’s algorithm and A* do not take vehicle constraints into account when planning. The path found is dependent on the graph used. There is however an adaptation of A* that can take vehicle constraints into account. It is called Hybrid A* and differs from the normal A*
algorithm by requiring the graph to be a grid, and allowing the solution to deviate from the graph nodes [4]. It still takes advantage of the heuristics just like A* which results in fast exploration towards the goal.

### 3.2.2 RRT*

The rapidly exploring random tree (RRT) was presented by Steven M LaValle in 1998 [13]. It is a path planning algorithm based on random sampling. It first adds a node in the start location and then randomizes a point in the free space and tries to extend the existing node towards that location. If the point is close it extends all the way to the new position and adds a new node there, otherwise it just extends part of the way before adding the new node. It then randomizes a new point and tries to extend the closest existing node towards that point in the same way as before. This results in a tree rooted in the start node. This process is visualized in figure 3.1. Every once in a while the goal position is picked as the random sample to ensure that the tree eventually reaches the goal, and once it does the algorithm terminates.

RRT* is an extension of RRT presented in 2011 by Sertac Karaman and Emilio Frazzoli [12]. RRT* works similarly to RRT except it doesn’t stop when the goal is reached and it also rewire the tree around new nodes every time a new node is added. During the rewire phase, edges between the nodes are changed if it improves the path from the start to that node. This means that as the nodes increase in number the path found goes towards the global minimum. The authors proves that the algorithm is asymptotically optimal, meaning that the algorithm will find the optimal path if such a path exists when the number of nodes goes towards infinity.


Algorithm 1 RRT

Require: start, goal

Tree ← start

while True do
    r = random(freespace)
    c = closestNode(Tree, r)
    N = steer(c, r)
    Tree ← N
    if N == goal then
        return N
    end if
end while

Figure 3.1: Step by step illustration of RRT
Algorithm 2 RRT*

Require: start, goal
GoalFound ← False
Tree ← start
while not terminated do
    r = random(freespace)
    c = closestNode(Tree, r)
    N = steer(c, r)
    near = CloseNodes(Tree, N)
    ChooseBestParent(near, N)
    rewire(Tree, N, c)
    Tree ← N
    if N == goal then
        GoalFound ← True
    end if
end while

RRT* FN

RRT* Fixed Nodes (FN) works the same way as RRT* does except it has a limit on the number of nodes in the tree [7]. Firstly the algorithm grows the tree just like RRT* but when the maximum node number is reached the algorithm removes a weak node before it adds a new one, thus keeping the number of nodes fixed. A weak node is defined as a node with one or no child.

RRT* with sparse tree heuristics

RRT* with sparse tree heuristics uses sparse sampling when building the tree to speed up the time it takes to find a path [8]. The sparse heuristics speeds up the exploration of the free space which leads to a faster discovery of a feasible path.

3.2.3 Distributed RRT

There have been several attempts at creating a true parallel implementation of the RRT algorithm, however it is difficult since the tree is built incrementally.
OR parallelization

OR parallelization is suitable for random algorithms. It solves the same problem on all workers at the same time, without any communication until one worker has found a solution. When a worker has found a solution it notifies the rest to stop. It might seem a bit unintuitive that trying to solve the same problem in parallel like this would result in any performance increase, but since the execution time needed for a random algorithm is random, solving the same problem multiple times would yield different execution times, thus solving the same problem in parallel would result in a shorter execution time in some cases.[3]

Environment decomposition

In the paper *A Scalable Distributed RRT for Motion Planning* [11] the authors presents a distributed solution for RRT which divides the search area between the workers to allow for local computations while keeping the communication between the workers low. The area is split into pie shapes and distributed among the workers (see figure 3.2). Furthermore the workers were only allowed to communicate with the closest neighboring workers to keep the communication low. This scaled well according to the authors.
MapReduce

The map-reduce framework was developed for large distributed computers. It requires that the computations are written in a specific way in order to guarantee scalability. The idea behind this is to divide everything into two steps, mapping and reducing. During the mapping stage the work is divided among the workers, which then does the requires computations and sends the results to the second step which is called reduce. During the reduce step the results from the workers are combined to form the final solution.

This method can not be used to parallelize the entire RRT algorithm, but it can be used for some of the heaviest computations. In the article PaRRT: Parallel Rapidly Exploring Random Tree (RRT) Based on MapReduce [1] the authors present such a solution where the nearest neighbor search was done with MapReduce. This resulted in a good performance gain on large datasets according to the authors.

3.2.4 Potential field methods

Potential field methods uses potential fields to find a path [2]. The potential field is usually added over the entire search space sloping towards the goal and increasing around obstacles (see figure 3.3). This
way a path to the goal can be found from any point in space by using gradient descent. The advantage of these types of methods is that they are usually fairly easy to compute which makes them usable in real-time, and it is easy to add additional temporary potential to avoid moving obstacles. They can however get stuck in local minimums which can be a problem in complex environments.

**Autonomous sailing**

The directions a sailboat can travel is influenced by the wind direction, so in order to plan a path for such a vessel the wind must be taken into account. This could be done in an algorithm like RRT* if such logic is included in the way the tree is built, however it might not always be the best idea since the wind direction tends to change especially in an environment with a lot of islands, which would require the path to be re-planned. To get around this problem a reactive algorithm built on potential fields could be used, either by itself or as a way of following a rough path planned by another algorithm.

In 2011 a reactive sailing algorithm was presented by Clément Pêtrès, Miguel-Angel Romero-Ramirez and Frédéric Plumet [10]. They created a potential field algorithm with several potential fields. One slopping towards the goal to bring the boat towards it, another around the boat with high potential into and away from the wind to prevent the boat from traveling directly into and with the wind, and a third one to
prevent the boat from tacking or gybing too often. This last field was added on the starboard side the the wind was coming from the port side and vice versa. By tuning the scale parameters for the fields the authors were able to get a sailboat to sail autonomously to a goal in changing wind conditions.
Chapter 4

Method

In this chapter the chosen approach is presented and explained.

4.1 Software architecture

The software architecture chosen was a block design. It is built up out of blocks to enable for easy maintainability of the code base, and to simplify adaptations to other vehicles in the future. This architecture is inspired by the one presented in [14], but simplified to fit the problem at hand. A block diagram with the different components is shown in figure 4.1.

An architecture like this also makes it easier to add and remove blocks if needed, it also makes it possible to share parts of the code between vehicles easily. The path planner for example could be the same in several vehicles if no vehicle constraints are needed.

![Figure 4.1: Overview of the software architecture](image-url)
4.2 Logging

An essential function of the software is the ability to log sensor data for post-processing and debugging. The format for the logs was chosen to be plain text. The reason for this was to make it easy to read without the need for decoding software. To further improve the readability of the logs, each block has its own file to write data into. This makes it possible to have different logging formats for different types of data if this is needed in the future.

4.3 Communication with sensors and actuators

The communication with sensors and actuators are done via a RS422 serial bus. The protocol used is a multidrop protocol with one master (the main micro-controller) and several slaves (the other micro-controllers). The slaves are connected to sensors and actuators. The slave nodes listens on the serial bus and replies to requests from the master. Figure 4.2 shows a simple example of the hardware layout.

Figure 4.2: Example of the hardware layout
The communication to the slaves can differ quite a bit between different sensors and actuators, for that reason a higher level interface is needed in order to keep the code easy to maintain. The underlying data transport protocol is kept the same, but instead of using it directly from the main part of the code a driver like object is used. One driver handles all communication with one slave and is responsible for asking for sensor data and sending set-points for actuators without any external function calls to do so, except an update function which is called regularly. This approach has the added benefit of being modular. A driver and corresponding slave can in most cases be added or removed without any changes to other parts of the code.

On the slave side, the micro-controller connected to a the sensor or actuator gathers data continuously, filters it if needed and stores it so it can respond quickly when data is requested by the master. It is also responsible for making sure the sensor operates as it should, meaning that if the sensor stops working, actions should be taken to try to make it work again, or notify the master if that isn’t possible. In an early state, the detection of a malfunctioning or broken sensor will be done by checking if it outputs any data, but more sophisticated methods could be used in the future where the data is analyzed as well.

### 4.4 Sensors

Any sensors capable of interfacing with the slave micro-controllers can be added to the system, there are however limitations on the amount of data the system is able to handle without slowing down considerably, so sensors like multi-beam sonars should instead be connected to the more powerful research computer on board of the vehicle in order to log and make use of the data. There are however some sensors that are needed by the system in order for it to be able to perform basic navigation. This includes sensors like Attitude and heading reference system (AHRS) and Global positioning system (GPS) to be able to determine the vehicle pose and position.
4.5 Localization

The position and pose of the vehicle are determined from GPS data together with an Attitude-heading-reference-system (AHRS). The underwater vehicle will also be equipped with a Doppler Velocity Log (DVL) to be able to measure speed when no GPS data is available. The DVL uses Doppler shift in sound waves to calculate the speed the vehicle travels over the ground. This information is combined with data from the AHRS to do dead reckoning.

4.6 Missions

In order for the vehicle to operate on its own, a way of storing and executing tasks is needed. This is done by a module called task-scheduler. This module has a list of tasks which it executes sequentially. The kind of tasks that can be added is, e.g, go to a location or follow a course.

4.7 Planning

The planning is needed when the vehicle has to travel to a location and it isn’t able to go there in a straight line. It is done in two steps. First a rough path is planned with a path-planner, then that path is followed by a reactive algorithm in real-time. This allows for the use of a path planner instead of a much more complex motion planner, and in addition to that it also allows the path planner to be kept the same regardless of the vessel.

The path planning algorithm chosen was an adaptation of RRT* combined with several simplifications and optimizations. Among the adaptations were sparse sampling to speed up the expansion of the tree, and a less accurate but faster way of finding nearest neighbors based on $[L^1]$ distance and two sorted lists. This simplification was not desired but necessary since the nearest neighbor search is the most time consuming operation by far otherwise.

$[L^1]$ distance also called Manhattan distance is the sum of the vertical and horizontal distances. $dist = |x_1 - x_2| + |y_1 - y_2|$
Algorithm 3 RRT* - Simplified nearest neighbor(s)

Require: \( P \) - position
Require: \( listA \) - list of nodes sorted on latitude
Require: \( listB \) - list of nodes sorted on longitude

\[ S_1 \leftarrow \text{findClosest}(listA, P) \] //Binary search
\[ S_2 \leftarrow \text{findClosest}(listB, P) \] //Binary search

\[ \text{return } \text{minRealDistance}(P, S_1, S_2) \]

The reason RRT* was chosen over a simpler algorithm like A* was the environment the vehicles will operate in and the possibility to add environment information like weather in the planning phase. In order to do the same with A*, the Hybrid A* approach explained in section 3.2.1 would have to be used. That algorithm does however require that the search space is discretized into a grid. The grid size would be chosen to fulfill the accuracy requirements needed and this can be a problem since different accuracies are needed at different places, which often results in a fine grid, and this is simply not possible to handle on a micro-controller due to the memory needed. RRT* on the other hand can create a path with both high and low resolution based on the need of the problem.

### 4.7.1 Maps of the free-space

A way of handling maps is needed for the planning. These maps needs to be as small as possible in terms of memory use. This excludes grid based maps. The format chosen for the maps is to use polygons defined by a series of points. One boundary polygon represents the outer boundary of the map, and other polygons represents obstacles and no-go zones.

The boundary polygon is not always necessary to have, but it is useful when generating random points for the RRT algorithm. In order to get a fast exploration with the tree, uniform distribution of the random points is a good idea. This can be done easily when a boundary polygon exits. The boundary polygon is divided into triangles and when a random point is needed one of the triangles is selected using.
weighted random selection with it’s area as weight. This means that a large triangle will be picked more often. A point is then picked at random within that triangle. Another approach of doing this would be to just randomize points all over the place and discard those which are outside the boundary. However, this is much less efficient due to the time it takes to check if the point is inside the boundary. If no boundary polygon exists, then this approach works well.

4.8 Distributed planning

A distributed approach for path planning is considered as an alternative to doing all planning on the master node. The approach chosen for distributed planning was the OR parallel approach explained in section 3.2.3. The reason this approach was chosen was because of it’s simplicity and the low demand on communication between workers.

4.9 Command filtering

A command is filtered before it is executed. This simply done by ignoring entire commands that are not allowed, or changing it so it becomes allowed. For example if a higher speed than what is allowed is requested, the speed will be reduced to the maximum.
Chapter 5
Implementation

In this chapter the software implementation and the underlying hardware are described.

5.1 The captain - network of TEENSY and ARDUINO

The main micro-controller used is a TEENSY 3.6, it is based on a the MK66FX1M0VMD18 arm core with a clock speed of a maximum of 180MHz and has 256Kb of RAM. This is very powerful for a micro-controller but almost nothing compared to a desktop computer. The micro-controllers connected to sensors and actuators are either based on the same processor or an atmega328P (Arduino UNO). The code is written in C or C++ and is run directly on the micro-controllers without an operating system in order to minimize overhead, and get as much performance from the system as possible.

5.1.1 Communication

The ways to communicate with the software program are limited to text based RF-communication and Radio control. The radio control is implemented in such a way that if the operator turns it on, he or she automatically takes control of the vehicle. RF communication is used for setting up missions and monitoring the vehicle while it is operating.
5.2 Surface vehicle - MARIBOT VANE

MARIBOT VANE is an autonomous unmanned sailboat developed at KTH. The purpose of the boat is to be able to collect scientific data from the oceans during an extended period of time. It is equipped with the electronics described in section 5.1.

5.2.1 Differences from a traditional sailboat

MARIBOT VANE uses a solid sail in contrast to sails made of fabric used by traditional sailboats. The solid sail on VANE has more in common with an aircraft wing than a traditional sail, it generates lift when the air flows past it. The amount of lift generated is controlled by a flap located behind the wing. When the flap is turned, it changes the angle of attack of the wing which in turn changes the amount of lift generated. The reason for using this type of sail compared to a traditional one is a higher robustness and easier to trim. A drawback however is that it is not possible to generate any useful lift when sailing downwind, for that reason the boat has to zig-zag just like any sailboat has to do when going upwind when it is going downwind as well.
To control the wing a simple set of rules are applied. If the wind is coming from the port side of the boat the flap is set to a positive angle, and the opposite is done if the wind is coming from starboard. This ensures that the lift generated by the wing pushes the boat forward.

5.2.2 Sensors

So far the vehicle is only equipped with the necessary sensors needed for navigation, this includes a GPS for determining the location, two
AHRS devices, one for the hull and one for the sail and an anemometer for measuring the wind speed and direction.

5.2.3 Low level path planning

Since MARIBOT VANE is a sailboat, the directions it is able to travel are limited by the wind direction. For that reason a special algorithm is needed to follow a path planned by the path-planner. The approach chosen for this is explained in section 3.2.4. Some simplifications had to be done compared to the approach originally presented by Clément Pêtrès, Miguel-Angel Romero-Ramirez and Frédéric Plumet [10] in order to make it work on the micro-controller. Firstly the full potential field is never calculated, instead the potential field values are calculated on a circle with a radius of 10m around the boat, then the direction corresponding to the lowest value is selected and set as the target course to follow. An illustration of the different components and the resulting potential field can be seen in Figure 5.2

This reactive algorithm could also be used without the path planner, and it would probably reduce the time it takes to reach a target location, however the reason the high level path is used is because of the fundamental problem with potential field methods with simple potential fields, it can get stuck in local minimums.
5.3 Underwater vehicle - MARIBOT LOLO

MARIBOT LOLO is an autonomous underwater vehicle under construction at KTH. Its main purpose is to act as a test and demonstration platform, and to do data collection under water. It will not be fully operational during the scope of this project and therefore tests with this vehicle is limited to surface operations only.

5.3.1 Sensors and actuators

MARIBOT LOLO will only be capable of surface operations during this project, and only equipped with sensors needed for that type of operation. It is equipped with a GPS and an AHRS. For propulsion
it is equipped with two electric motors. The motors will also be used for steering because no rudders would be operational during the first tests.
Chapter 6

Experiments

In this section the experiments are explained. The purpose of the experiments is to investigate if the proposed software is capable of controlling an autonomous vehicle as intended, and to determine if using more than one micro-controller in the path planning process increases the performance.

The performance of the control is evaluated by giving the vehicle a mission to travel from one location to another. If it is able to do so without the need for an operator to intervene, it is considered to be working.

The performance of the path planning is measured in terms of if the planning was successful, meaning that a path was found, and the time needed to find the solution. Since the OR parallelization technique executes the same algorithm on all nodes at the same time, and the computation is terminated as soon as a solution is found, the solution generated by the distributed planner and the local planner can be assumed to be of the same quality, for that reason the length of the path is not used as a performance indicator, but a few example paths will be shown.

6.1 Simulated tests

Simulated tests are used for determining the performance of the planning algorithms. Maps of different sizes and complexity from the Stockholm archipelago are used to get an idea of how the performance
would be in a real life scenario. The maps can be seen in Figure 6.1.

The tests are carried out on a purpose built system consisting of a mas-
ter node and three worker nodes. The test process is as follows. A map
is loaded on the master and sent to the workers. Two positions on the
map are randomized, one as the starting point and another as the goal.
The goal bias is also randomized as a value between 50% and 12.5%.
The goal bias determines how often the RRT algorithm should try to
extend to the goal. A high value of 50% means that the algorithm will
try to extend to the goal every other iteration. A high goal bias is ex-
pected to be good in an environment that is mostly free, because the
tree will extend towards the goal faster, however in a complex map a
lower goal bias might yield better performance, since it allows the tree
to expand more. Once the start, stop and goal bias has been generated,
the distributed path-planner is used to plan a path and the execution
time is recorded, once it is done, the same planning is done locally and
the results recorded in a file.

In order to get a good picture of how factors like distance and goal bias
influences the performance between 15000 and 20000 tests are done on
four different maps.
Figure 6.1: The maps used for evaluating the performance of the path-planning
6.2 Sea trials

Most of the sea trials were carried out with the MARIBOT VANE, since this vehicle has come further in the development and is easier to test. The software was tested in the AUV LOLO as well but just briefly on the surface.

The sea trials done with VANE were designed to test the autonomous sailing algorithm together with the path planning and also to collect data about the boat’s performance in a variety of wind conditions.

The trials were carried out in the Stockholm archipelago. The vehicle was driven with an RC controller to get into open water, once there RF communication was used to set up a mission, which then was started. During the trials, the vehicle was monitored and RC was used to take control if there were any risks of collisions with other vehicles.

Only brief tests were carried out with MARIBOT LOLO during this project. The tests included launching the vehicle, towing it to open water and trying to follow a course for a short period of time, before recovering the vehicle again. No complex functionality like the path planning could be tested during this trial, but it was possible to evaluate how the system worked by looking at the logs.
Chapter 7

Results

In this chapter the results from the simulated tests and sea trials explained in Chapter 6 are presented.

7.1 Simulated tests

Figure 7.1 shows examples of the paths planned by the path-planner on different maps. Figure 7.2 shows a section of the path on map (d). The blue lines show the tree generated by the path planning algorithm, and the red line which is the path chosen. The red markers show the positions that will be used as targets for the low level planner when it is following the path.
Figure 7.1: Example of paths planned on different maps together with the tree built

Figure 7.2: Section of the path and tree
Figure 7.3 shows how the time needed for planning a path is affected by the number of workers and the goal bias in general. The results come from the tests on map (d). Due to the randomness of the algorithm, the time needed is also random to some degree, it still changes with distance since more nodes are needed but a long distance between start and goal does not necessarily take longer time than planning between to positions closer to each other. The mean and standard deviation for the local planner and the distributed planner with three nodes is shown in figure 7.4. The solid line shows the mean, and the colored area shows the standard deviation.

Figure 7.5 shows how much the additional nodes speeds up the planning.
Figure 7.3: Expected time needed to plan a path of a given length with 1, 2 or 3 workers and different goal bias settings.

Figure 7.4: Mean and standard deviation of computation time for local planner and distributed planner with three nodes.
Figure 7.5: Speed increase with different number of nodes on map (d)

Figure 7.6 Shows success rate of the local and distributed planner on all four maps.
7.2 Sea trials

In this section the results from the sea trials are presented

7.2.1 MARIBOT VANE

The track of two of the sea trials with MARIBOT VANE is shown in figure 7.7. The left image shows the first test, the vehicle sailed autonomously for about four hours with just minor interference from the operators. During this trial a problem with the way the wind angle and speed was filtered was found, which led to the less than ideal track seen in the upper right corner of the image. Another problem found during this trial was a problem with the GPS receiver which stopped working, and had to be restarted manually by entering a command via the RF link.

The second trial shown in the right image was designed to see if the problems found with the filtering of the wind was sorted, and to collect data about the vehicle’s performance when sailing in different angles relative to the wind. This was the longest trial at this point and
lasted about 8 hours. The autonomous sailing worked much better this time, but there were still some problems with the filtering. The GPS issue which was thought to be solved at this time still persisted. A closer image of how the boat follows the path planned can be seen in figure 7.8.
7.2.2 MARIBOT LOLO

Two tests were done with maribot LOLO during this thesis. The tests were the first times the vehicle was in the water and focus was on making sure the vehicle behaved as expected rather than testing features in the control software. The vehicle was not fitted with all the intended hardware.

Figure 7.8: Section of the track seen in (b)

Figure 7.9: Track from maribot lolo
Chapter 8

Discussion

The software proposed is able to handle most of the tasks required. Both vehicles tested shares most of the software, the main differences between them is in the logging of variables, controllers and the low level path planning. It was easy to adapt the software from VANE to LOLO when it was time to do sea trials with that vehicle.

Some of the goals were not met due to time constraints, e.g the return to home if something fails. All the logic for traveling back to a starting point exists, however a way of detecting the error has not been implemented.

During the sea trials with VANE it became clear that having software that monitors the sensors and restarts them if required is important, this was something that was left out due to time constrains before those trials, but later added for the LOLO trial.

While the software could be tested extensively during the VANE trials, the LOLO trials did not allow for that. This was expected since the LOLO was far from being finished in contrast to VANE which was been launched the previous year and more than 10 shorter trials were done throughout the summer when features were developed and added to the software.
8.1 Autonomous sailing

The real-time algorithm which follows the path planned by the high level planner in MARIBOT VANE worked well. It outperformed the expectations and was not only able to sail to a location in any wind angle, but it was also robust enough to reach the target location when the wind angle was calculated incorrectly.

8.2 Path planning

It is possible to plan a path for a vehicle on water with the algorithms discussed in this thesis for a system of distributed micro-controllers.

The RRT algorithm used in this thesis might not always be the best choice for an application like this, but it seems to work and plans a fairly safe path in most cases. Other algorithms like using a graph search algorithm on a graph computed from the map, e.g. a visibility graph would result in a more optimal path in terms of distance, however the shortest path might not always be the best, for the sailboat for example traveling close to islands is bad because of the risk of running aground, and also if the vehicle is located downwind from an island, the island will prevent the wind from reaching the vehicle. There are of course other graphs to use like a Voronoi graph, but depending on the situation it might result in an even less optimal path compared to the one generated by RRT.

The main reason the RRT algorithm was considered in the first place was its ability to do motion planning, this did however not turn out to work well since all the extra computations needed made it very slow, however a mixed approach could be used with the RRT where a cost is added when expanding the tree for e.g wind and wave conditions to allow it to plan paths around bad weather if such data is available.
8.3 Performance gain in a distributed system

There was an increase in performance as seen in Figure 7.3 and Figure 7.5 when using the distributed approach for the path planning. Using more nodes in the planning reduces the time needed on average, moreover the standard deviation is reduced as seen in Figure 7.4. By using two nodes instead of one, the computational time reduced by an average of 27% and with three nodes 36%. The biggest reduction is seen on short paths, but its worth noting that the maps were transferred before the timing was started during the experiment.

The reason the distributed approach for path planning was first considered, was the potential increase in tree size by having decentralized storage of the tree. This approach was however deemed too hard to implement on the current system. The main limitation is the communication between the nodes. If two nodes has to share information with each other, that information must go through the master. This is far from ideal since the latencies would overshadow any potential performance increase. If such an approach were to be considered in the future, another network layout is suggested.

The performance gain by distributing computationally heavy tasks like path planning is not high enough to compensate for the extra complexity added, but it has other benefits. By moving the computations needed to do the path planning to another node, a rather large and complex part of the code can be removed from the master node. This reduces the risk of having fatal bugs in the main micro-controller code and thus increases the robustness of the system. If the node doing the path planning would freeze, the rest of the system will still continue to operate and communications with the operator would not be affected.

The main potential performance gain by utilizing a distributed system comes from local processing of sensor data before it is sent to the main micro-controller. The amount of this gain was not measurable on the available systems, since none of the micro-controllers has enough inputs to connect to all sensors at the same time. The performance gain would most likely only be noticeable when extensive local processing
is done on the sensor data.

8.4 Validity of the Results

The results of the field tests aimed at answering the research question: *Can distributed micro-controllers be used to control an autonomous vessel?* are clearly valid since several successful sea trials were conducted with the system, thus proving that it is possible. It should however be noted that the environmental conditions such as wind were not controlled and did change during the trials. However, the software was developed to work in a real world scenario so changing conditions should not affect it’s ability to operate.

The tests done for answering the continuation of that question *And is it possible to solve complex tasks like path planning on that hardware?* clearly show that it is possible to some extent. During the development of the method several simplifications had to be done in order to get a decent performance, so while it is possible the solutions it generates does most likely not achieve the same quality as it could have if a more powerful system was used.

The tests done for answering the second question about the performance gains by using a distributed approach shows that it is possible, and it increases performance slightly, especially in terms of time needed to find a solution. The tests conducted did however not include the time it takes to set up the computation nodes with the needed information such as a map of the environment. If that was to be included as well the performance difference would most likely be smaller. There is also no investigation done to how the normal tasks of the computation node e.g processing data from sensors are affected by the extra time spent on computing a path. This is difficult to measure, especially on a purpose built system without sensors, which was used during these tests. While this system was built to work as the real system in the vehicles, it is not exactly the same and tests done on the real system while it is operating might differ slightly.
8.5 Ethical and social aspects

The ethical aspects of this project could include surveillance and integrity since an autonomous vehicle can be used to collect data about people unknowingly. The work in this thesis does not enable any such surveillance by itself, but it could be used as a starting point for someone else. Another consideration could be the loss of jobs when autonomous vehicles are used instead of manned vessels, however the vehicles this thesis is aimed for are small underwater vehicles and small surface vessels which are too small to be crewed.

From a broader perspective the findings in this thesis could lead to a realization that cheap hardware can be used in autonomous vehicles without sacrificing features. It also means that vehicles thought to be outdated could continue to operate with just minor inexpensive changes to hardware, which can lead to a more sustainable and economical operation.

8.6 Future work

Extending the path planning algorithm to take environmental data into account such as currents and wind. This enables the vehicle to avoid unfavorable weather that could risk harming the vehicle.

An other extension for the sailboat would be to use the Automatic Identification System (AIS) to get information about surrounding boats and add them as obstacles during the autonomous sailing algorithm. This would make it possible to avoid other boats and reduce the risk of collisions.
Bibliography


