Autonomous Sailing Robots
Project Report

Martin Endreß, Tobias Heckel, Ostap Kharysh, Yaryna Korduba, Ralf Lederer, Alexander Löflath, Jan Martin

Smart Environments
Faculty of Information Systems and Applied Computer Sciences

Abstract. This project report describes the results of the smart environments project „autonomous sailing robot“ which was executed in the winter semester 2017/18 at the University of Bamberg. The project's goal was to develop an autonomous sailing robot which can be used as a mobile sensor platform. Our solution provides a middleware, a navigation algorithm, a motion controller and a hardware simulation. As a basis, we used a modified, commercially available, remote controlled sailing boat.

Keywords: robotic sailing, autonomous sailing, middleware

Table of Contents

1 Introduction

Autonomous sailing robots provide new opportunities for ocean observation, e. g. intelligent sensor buoys or mobile sensor platforms for surveillance of environmental parameters or mapping of minefields (?, ?).
We provide a solution for an autonomous sailing robot consisting of three parts. The section *Middleware* describes the interface between the boat's hardware and higher-level subsystems. Furthermore, a *Simulation* is introduced which can substitute the real hardware for test purposes.
Section three and four describes the higher-level subsystems, i. e. the *Navigation* that is responsible for finding a route from a start to a goal coordinate and the *Motion Controller* interprets the data delivered by the navigation and the middleware and controls the actuators of the boat.
The fifth section concludes the project report.

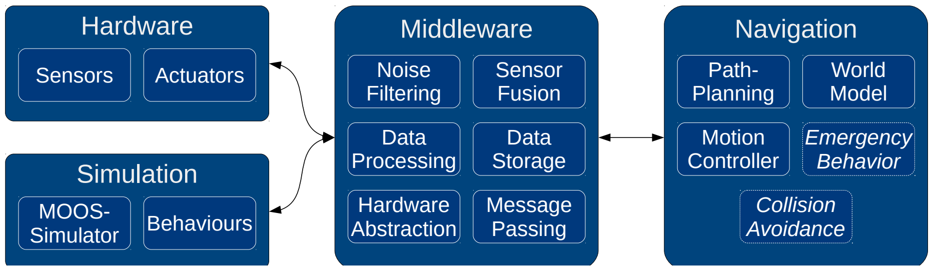


Fig. 1. The high-level-architecture

2 Middleware

The Middleware is responsible for connecting higher level subsystems, such as the navigation system, to the sailing robot. Therefore it abstracts the hardware of the robot to a level where high-level tasks can efficiently be performed. In order to do that the Middleware provides a central data accessing interface which can be used to retrieve data about the sailing robots current state and request state changes in terms of setting the rudder and the sail position of the sailing robot. Thus the Middleware communicates with the boats hardware to perform sensor readings. It filters, stores and processes this data.

2.1 Initial decisions

The team, responsible for designing and implementing the Middleware, consisted of three members. Ostap Kharysh, Yaryna Korduba and Ralf Lederer. To guide our implementation work, we first decided to implement the Middleware as a network of adjacent, but independent subsystems. Each node in the network has its own specific purpose. By using this architecture the robustness of the system is increased. If one node fails, the others can still perform and keep the network running. Further more the computational power needed to keep the the network running can be spread over multiple computer systems. One could also think about backup systems and nodes that participate in the network if needed. In order to evaluate software parts of the sailing robot like the navigation system, it is necessary to observe the boats behavior according to the purposed navigation tasks. Doing this with the real boat and under real-world conditions is very time intensive and expensive since the environmental parameters, like the wind strength and the wind direction can't be controlled easily. Therefore we decided to use a simulator to delay testing in the real environment to late project phases. By using a shared architecture a simulator-node can easily be attached to the Middleware-network. Yaryna Korduba focused on finding and connecting a suitable simulator. In order to share data, nodes must be able to communicate with each other. Therefore they need to make use of a transmission and a communication protocol. One node was specified as central data accessing point, namely the blackboard node. It uses the blackboard architecture and keeps the central information of the sailing robots current state. Other nodes are able to retrieve data from the blackboard and post data to it as well. Furthermore the blackboard is the data accessing interface for higher-level subsystems.

After researching about the robot operating System ROS (?, ?), which is used by the university of Southampton to run a similar project (?, ?), we opted to implement our own JAVA-based Middleware to meet our time constraints. The software design and the JAVA implementation was done by Ralf Lederer. Ostap Kharysh focused on researching and implementing noise filtering and sensor fusion using different kalman filters.

2.2 Architecture

The Middleware represents one layer in the layered architecture of the sailing robots software system. On the one side it is connected to the boats hardware, on the other side it is connected to systems that perform higher-level tasks, like the navigation. The Middleware itself consists of a network of **nodes**, each responsible for a specific task. **Nodes** share data and communicate with each other. Thus a transmission system is required that provides the interchange of data. This includes specific transmitting techniques as well as a communication protocol which specifies the syntax and the semantics of the exchanged data. Multiple nodes can be together hosted in a single software process or each node in

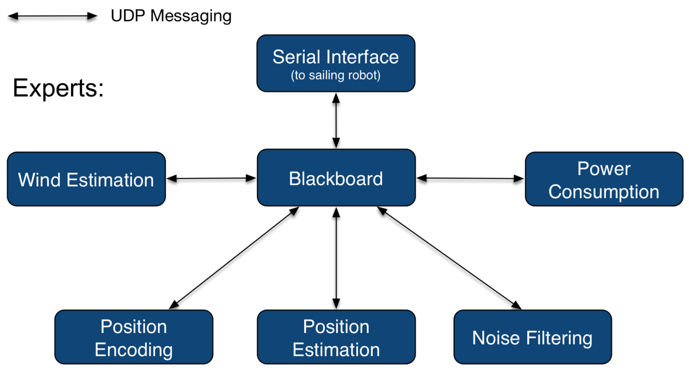


Fig. 2. nodes of the Middleware

its own one, e.g. when a specific node runs on another computer system to spread needed computing power. The network of nodes representing the Middleware is shown in Figure 2.

2.3 Implementation

2.3.1 Communication between nodes

The communication between nodes is provided by command sources. A command source is an abstract object which is able to send and receive control commands. It abstracts from the underlying transmission technique used for the communication. Control commands are data strings that correspond to the communication protocol and they can have different purposes, such as the requesting and sending of data or subscribing to specific sensor readings. Once received by a command source, control commands are interpreted by an interpreter which determines the purpose of the command and checks if the syntax of the command is valid. To react to a received command, a command source can be observed by arbitrary many command source listeners. These listeners get informed if a command was received to decide how the command shall be processed. A command event listener does not need to observe all kinds of received commands. It can observe only commands with a specific function, like data deliveries for a GPS sensor. To enable the communication between nodes, all nodes use at least one command source to send and receive data. If a command source receives a request for which it has no listener, the command will not be processed. A communication process between two nodes is visualized in Figure 3.

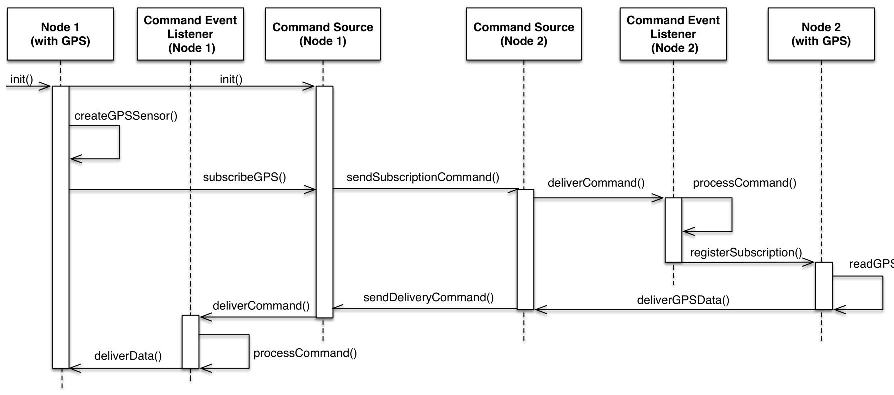


Fig. 3. subscription & data delivery process between two nodes

The abstract class command source provides basic functionality suitable for all command sources. Basic functionality includes the registering and informing of event listeners. Sending and receiving needs to be implemented by concrete command sources and depend on the transmitting technique used. In our case we have two concrete implementations for command sources. One for the communication between nodes over an ip-based network, and one for the communication between the Middleware and the boats hardware over a serial interface.

2.3.2 Communication protocol

The communication protocol specifies the syntax and semantics of the control commands sent between command sources and thus nodes. It is independent of the underlying transmission technique. Three types of actions were specified.

1. Queries

Queries are used to request data and have the following form:

$?type()$

where type denotes the function of the sensor and thus of the data to be queried, e.g. $?gps()$ queries the current GPS sensor reading.

2. Continuous queries

Continuous queries are used to subscribe for data deliveries of a specific sensor and have the following form:

$?type(?)$

The question mark in the brackets tells the system to send a delivery to the requesting command source every time there is a new sensor reading for the sensor having the function denoted in type.

3. Deliveries

Deliveries are used to deliver data from one node to another. Deliveries can be an answer to a previously received query or be sent as an answer to a subscription. Deliveries have the following form:

$type(timestamp, val1, val2, val3, \dots)$

where the order and the amount of the values depend on the type. The timestamp denotes the time when the sensor reading was performed by a sensor or created by a virtual sensor. For example $gps(400, 5306.3280, N, 851.2623, E, 0.00, 212.30)$ delivers the current GPS reading of the sailing robot. Deliveries can also be used to send commands to control the boats actuators, e.g. $rudder(30)$ tells the sailing robot to move its rudder to 30 degrees.

All characters, as well as numbers, are transmitted by using ASCI strings. This produces more network load than using byte representations, especially for the numbers. However, it was decided that a human readable communication protocol would be better, especially for validation and testing purposes. If the system will be extended in the future, the communication protocol may need to be redesigned to optimize the message load if needed.

The following table contains all commands for sensors and actuators related to the boats hardware which were implemented during the projects execution. Commands marked with a (*) can only be interpreted by the boats hardware, but not by the Middleware:

function	command string	ranges	explanation
gps	gps(timestamp, long, direction_long, lat, direction_lat, speed, heading)	According to NMEA 0183	transmits the current gps reading
imu	imu(timestamp, accelerometer_x, accelerometer_y, accelerometer_z, gyro_x, gyro_y, gyro_z, compass_x, compass_y, compass_z)	See sensor specification (SparkFun 9DOF - Sensor Stick)	transmits the current imu reading
imu(*)	imu(rate, n)	any integer	sets the data rate of sending imu messages to send only every n seconds
imu	imu(readings, n_acc, n_gyro, n_compass)	any integer	gives the number of readings currently accumulated in the internal buffer (which is reset whenever an imu message is sent) for all three sensors of the imu module
imu	imu(errors, n_acc, n_gyro, n_compass)	any integer	gives the number of communication errors with the respective sensors since the last query for errors
windvane	wind-vane(timestamp, angle)	0 to 360 degrees	not yet supported
rudder	rudder(timestamp, angle)	-90 to 90 degrees	sets the standing of the rudder
sheet	sheet(timestamp, angle)	0 to 100 degrees	sets the standing of the sheet
light(*)	light(parameter)	prameter = on, off	turns the attached LED's on or off

light(*)	light(blink, pattern)	see explanation	sets blink pattern for lamps, where pattern is a comma-separated string. Each component is shown for 100ms, when the end of the pattern is reached, it is started again. In each component, "R" turns on the red lamp, "G" the green, "H" the white. Example: "RG„W„," means Red+Green at step 1, all off at step 2, White on at step 3, all off steps 4-6.
voltage	voltage(timestamp, x)	any float	gives the battery voltage in millivolts (the system runs on 6 NiMh cells)
current	current(timestamp, x)	any float	gives the current in mA (roughly 30mA with servos/lights off)
consumption	consump-tion(timstamp, x)	any float	gives the total energy consumption since system startup in mAh (around 3000mAh of power are available in a freshly charged battery)
temp	temp(timestamp, x)	any float	gives the temperature sensed by thermistor attached to the hull, i.e., water temperature
reset(*)	request a hardware reset of the central microcontroller. Peripheral hardware is not reset since GPS and IMU require a power cycle. The system will respond with RESET("reason") where reason is a string listing all flags indicated by the system's reset controller as reason for performing the reset. Possible reasons are "software" (triggered by requesting reset), "PDI" (triggered by external in-circuit programmer), brown-out (undervoltage), external (pushing reset button), power-on, and watchdog. The reason "watchdog" indicates a system hang which has triggered the internal watchdog circuitry		

The following table contains the commands for virutal sensors and sensors that have been implemented for simulation purposes:

function	command string	ranges
local_position	local_position(timestamp, x, y)	defined by mercator projection
gps_dec	gps_dec(timestamp, latitude, longitude) (note: for transmitting gps commands in decimal latitude/longitude format)	latitude: -90 to +90, longitude: -180 to +180
rotation	rotation(timestamp, x, y, z) (note: for transmitting the results of the sensor fusion estimating the rotation of the sailing boat)	+/- 0 to 360 degrees for all axes

Commands for the battery consumption were not implemented since there was not enough time within the project.

2.3.3 Transmission protocols

For transmitting data two transmission protocols are used.

USART

For the communication between the boats hardware (ATXMega) and the PC running the blackboard node, a serial communication is used. Therefore a concrete implementation of a command source was created which is able to send and receive control commands over a serial communication, namely the SerialCommandSource. All physical sensors attached to the blackboard are registered as command event listeners of the SerialCommandSource and are therefore able to process data from it. Every time the sailing robot sends new sensor readings through the wireless communication, the ATXMega redirects this data to the blackboard using the serial interface, connected via USB. Other nodes can then access the blackboard to use this data. Every time data is read from the serial communication, it is stored into a buffer. Commands sent by the ATXMega are separated by the separator character $\backslash n$. The buffer can then be processed by looking for the separator to separate and process single commands. Subscribing for serial command sources was not implemented during the project because it wasn't needed, but it could easily be added in the future. When the blackboard receives control commands for the rudder or the sail from other nodes, it needs to redirect these commands to the serial interface such that the sailing robot is informed about the new data. It can then set its actuators correspondingly. To establish the serial connection the JAVA library ? (?) was used.

UDP

The transmission protocol used for the communication between nodes in an ip-based network is the user datagram protocol (UDP). We decided to use UDP since it has several advantages over the TCP protocol for our purposes. UDP is not connection based and it is faster than TCP. Although UDP is less reliable in terms of that there is no guarantee that packages will arrive, we decided to use it because of its smaller overhead. Further more packages over UDP can be send in any order and don't need to arrive in the same order as they were sent. An error recovery, like implemented in TCP, is not needed because there is so much traffic in the network that the loss of single packages won't affect the reliability of the system. The command source, namely UDPCommandSource, implemented for the communication provides methods for sending and receiving UDP packets. For that each UDPCommandSource is supplied with a network address and a port under which it is reachable. All nodes hosted on the same machine are reachable over the localhost. When a UDP packet is received, the receiver can determine the origin of the packet by looking at the senders network address. This is important when a subscription or a data request command is received. For each subscription the connection information of the subscriber is stored in a list, together with the name of the subscribed function. With this information a command source can determine which subscribers shall be informed when it receives new data for a specific function. In detail the process works as follows: First a command source receives a data delivery command for a specific function, e.g. the GPS. It then checks if it has any subscribers for that function. If it has subscribers, it sends the delivery command to them.

2.3.4 Command validation

Commands received by a command source must be checked for validity. The interface command interpreter specifies which properties of a command need to be verified. A concrete implementation of a command interpreter then checks the validity based on the used communication protocol. Since we use only one communication protocol, only one concrete implementation for a command interpreter exists, namely the StandardCommandInterpreter. A command interpreter further more is responsible for parsing arguments and determining the number of arguments contained in a command. By using regular expressions the command interpreter checks if the syntax of the received command corresponds to the communication protocol.

Arguments of data delivery commands must be either numbers or the letters N, E, S or W. Parsing these letters is necessary for parsing cardinal points, e.g. N for north, E for east and so on. Number arguments are allowed in different number formats. Allowed are signed numbers, integers and floating point numbers, e.g. $+3$, -3.2343. Further more floating point numbers are allowed to have scientific number notation, e.g. $+3.2343E10$.

2.3.5 Sensor class

Sensors are software objects that are used to represent hardware sensors attached to the sailing robot. Each node holds instances of the sensors which it uses to operate. The blackboard node contains instances of all sensors. A sensor object implements the command event listener interface. By doing that it is able to update its values when the command source it is attached to receives new readings. On the other side the sensor can provide data if a command source requests it. Technically a sensor observes control commands which addresses the sensors function. A concrete implementation of a sensor implements the abstract class sensor and overrides the functions for providing a function name, a description and a specification of the sensors values. This sensor specific information is different for each concrete sensor implementation. The basic functionality for sensor implementations is summarized in the abstract class sensor. A sensor is able to create control commands according to the communication protocol for delivering, requesting or subscribing data. When a sensor receives new data it checks if the number and data types of the received data arguments match the sensors specification. If the data is valid it is stored by the sensor. If a sensor receives a reading request it directly responds to the senders command source by sending the corresponding data delivery command. The sensor always replies with its newest reading. A sensor is observable with the sensor observer interface. Observers get informed if the sensor received a new reading.

Virtual sensors

Virtual sensors describe sensors which are not present in hardware and are therefore virtual. They are created by the Middleware and are used by nodes that execute tasks like sensor filtering and sensor fusion. The output of a sensor fusion may for example be a position estimate of the sailing robot by combining the sensor readings of the GPS and the accelerometer. To use the produced outputs like a normal sensor, the estimated position is modeled as virtual sensor and can then be used like any other sensor.

Storing values

Sensors are able to store a list of their last readings, received by a command source. Because readings are transmitted via UDP it may happen that deliveries will be delayed due to network problems. Because of that it is necessary to time stamp control commands such that the receiver is able to determine when a reading originally was created. If control commands and thus sensor readings are received delayed the sensor is responsible for ordering them correctly. Values are stored in hashtables by using key-value pairs. The number of keys specifies the number of values a specific reading must contain. If a sensor then receives a reading with less values, it can detect that something is wrong with it. A received and processed reading is then stored in a list of up to 1000 sensor readings. The readings must be ordered regarding to their time stamp they were created, not by the time they were received. This is important when readings are used to inspect their evolution over time, for example by an integration over the forces applied to the accelerometer to determine the velocity between two time units.

For storing sensor readings we opted for using a single linked list of a fixed size because it provides low-cost operations for inserting elements in the end, and remove elements from the beginning. When a sensor reading is received and the list is full, the oldest reading is removed which is on the first position of the list, then the new reading is added to the end. When a reading was received delayed the correct position to insert must be found. Because we assume that this case will rarely occur, we decided to use a linked list instead of a skip-list which would be more efficient if many search operations must be performed. A visualization of an insert operation of a delayed reading is shown in Figure 4.

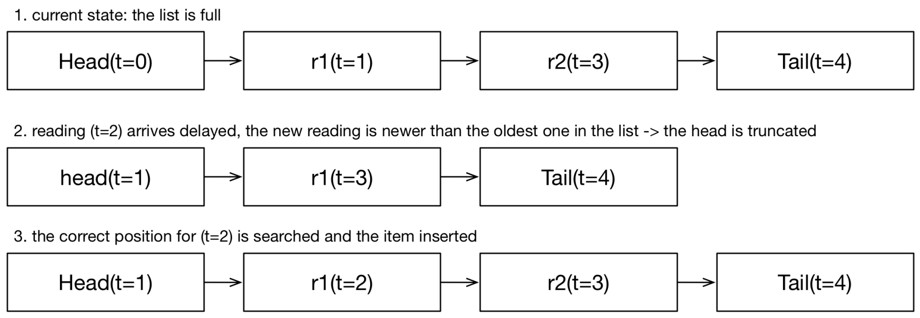

1. current state: the list is full
Head(t=0)
r1(t=1)
r2(t=3)
Tail(t=4)
2. reading (t=2) arrives delayed, the new reading is newer than the oldest one in the list -> the head is truncated
head(t=1)
r1(t=3)
Tail(t=4)
3. the correct position for (t=2) is searched and the item inserted
Head(t=1)
r1(t=2)
r2(t=3)
Tail(t=4)

Fig. 4. insert operation of a delayed sensor reading received by a command source

2.3.6 Experts

Experts are nodes that have a specific purpose. Usually experts access the blackboard to retrieve data, process the data and post it back to the blackboard. The blackboard is an expert for data storage and data management. The abstract class expert provides basic functionality for all experts. It contains a UPD command source and a list of sensors attached to the expert. When a sensor is added to the expert it is registered as command event listener to the experts command source. By that the sensor is able to get informed about received commands. Further more experts are able to execute scheduled tasks which are processed in their own thread repeated by a specific time interval. This feature is used to perform tasks like sensor fusion or noise filtering that apply in fixed time intervals. A specific expert implementation may override the working methods to implement expert specific behavior. By observing other experts, experts are able to get informed about new data for specific sensors. Usually the blackboard expert is observed by all other experts. If an experts creates new data it is responsible for time-stamping it. The applied timestamps correspond to the newest timestamp of the sensor readings which were used to create the new data. An expert may also provide a virtual sensor for new data which has no corresponding physical sensor. E.g. there is an expert for converting global position information from the gps sensor to a local coordinate system. Therefore it provides a virtual sensor for local position information.

2.4 Expert-Nodes

2.4.1 Blackboard

The Middleware provides a centralized data accessing node, namely the blackboard. The blackboard represents a dynamic data storage which can be read and written by other nodes. Every time the sailing robot sends new data to the Middleware, it is stored on the blackboard. Other nodes that work with this data can access the blackboard to read it. A typical task for a node could be a filtering process. Thus it reads data, performs some filtering action and then posts the new data back to the blackboard. Because nodes do not know when the blackboard receives new data which they need to operate, they need to perform readings in a fixed time interval, even if there is no new data available. To overcome this polling effect the blackboard provides a publish-subscribe mechanism. Nodes can register for specific kinds of data, e.g. readings of a specific sensor, and get informed by the blackboard if there is new data available. This reduces a lot of network overhead since no polling is needed anymore.

The Blackboard further more provides the data accessing interface to higher-level subsystems. Tasks can access the blackboard to read information about the boats current state. They can also send control commands to the blackboard which are then redirected to the boats hardware. Controllable actuators of the sailing robot are the rudder and the sail. Therefore the blackboard can be used to apply new angles for rudder and sail in terms of navigation purposes and current wind conditions.

2.4.2 Local Position Encoder

The local position encoder expert is used to convert gps readings (lat/long) to a local coordinate system (x,y). The global gps coordinate system is not well suitable for doing navigation tasks since data readings are in the NMEA-0183 data format. Therefore the converter firstly converts the gps data format to decimal (lat/long) coordinates. For the local coordinate system then a cylindrical mercator projection is used in which the world is modeled as a 2D surface splitted into multiple cells along the horizontal and the vertical axis. The origin (0,0) lays in the center of the projected 2D map. Because the earth is not a perfect sphere, the cylindrical projection is afflicted with distortion. The distortion is small near the equator regions and increases with the latitude from the equator to the pole regions. When performing local movements it is important to compensate the distortion created by the used projection when converting back local positions into a global coordinate system. For example: let the local coordinates (10000, 10000) be the current position of the sailing robot. When the navigation algorithm indicates the robot to make a turn at (10000, 10010), ie after 10 meters in y-direction, then 10 meters only apply in the local, but not in the global coordinate system. To get the correct movement in the global coordinate system Δy needs to be multiplied with the scaling factor k. The local position encoder subscribes the blackboard for the gps sensor. Every time the blackboard receives a new gps reading from the sailing robot, the position encoder is informed and converts that reading to a local position. It then posts the local position back to the blackboard such that other experts can access and use this information.

1. The (x,y)-coordinates are calculated as follows:

$$x = R \cdot \lambda^{\circ}$$

$$y = R \cdot \ln \left[\tan \left(\frac{\pi}{4} + \frac{\varphi^{\circ}}{2} \right) \right]$$

where R is the radius of the earth and λ° is the angle of the longitude in radians and φ° is the angle of the latitude in radians.

2. The inverse transformation (lat, long) is calculated as follows:

latitude: $\varphi^{\circ} = 2 \cdot \arctan \left[exp \left(\frac{y}{R} \right) \right] - \frac{\pi}{2}$
longitude: $\lambda^{\circ} = \frac{x}{R}$

where R is the radius of the earth. To get the coresponding coordinates in meters we used the value $R = 6378137.0$ which is the radius of the earth in meters.

3. The scaling factor k is calculated as follows: $k = cosh \frac{y}{R}$

For the conversion from NMEA-0183 to decimal (lat/long) the code from ? (?) was used. Because there were some errors in the code some corrections have been made. The workflow of coordinate conversion is shown in Figure 5.

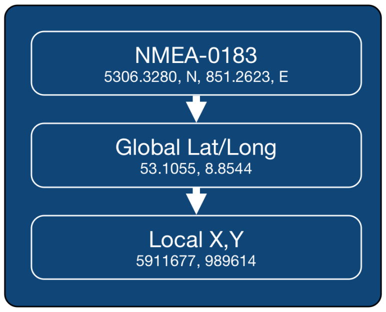


Fig. 5. workflow of position encoding

2.4.3 Wind Expert

The wind expert was intended for estimating the current environmental wind conditions, like current wind strength and wind direction. This information could have been used to predict the boats movement and make navigation algorithms more efficient. Unfortunately the ideas for such an expert where discarded because the used calculations weren't precise enough to result in a good estimate. Nevertheless the ideas and results shall be provided here.

A boat is always situated in the physical environment. Of course, there are several forces acting on it, which should be counted in order to conduct a successful sailing. At this part, closer look at this forces and possible calculations is suggested. As this project is about a sailing boat, all forces and calculations are explained from the sailing perspective.

To begin with, the are two basic parameters (two types of wind) we were interested in: True wind and Apparent wind. **True wind** is the wind relative to a fixed point the observation of which is not affected by the motion of the observer. **Apparent wind** the wind as observed aboard a moving vessel, being the vectorial combination of the true wind and the wind due to the ship's motion.

Relying on ? (?) there are several calculations that could be conducted to calculate angles and speed:

- Apparent wind speed: $A_s = \sqrt{T_s^2 + B_s^2 + 2T_sB_s\cos T_a}$, where T_s - true wind speed, B_s - boat speed, T_a - true wind angle to the object.
- Apparent wind angle: $A_a = \arccos((T_s\cos T_a + B_s)/A_s)$, in case $T_a > 180$ following equation should be subtracted from $360°$.
- True wind speed: $T_s = \sqrt{A_s^2 + B_s^2 - 2A_sB_s\cos A_a}$.
- True wind angle: $T_a = \arccos((A_s\cos A_a - B_s)/T_s)$.

Considering, above mentioned calculations of wind physics it is possible to move forward to the boat itself. Area of interest now is how to interpret right acceleration, and speed which could be expected from this forces acting on the sail. This part of calculations requires to know several default boat and environmental parameters:

- Maximal Hull speed of the boat: $H = 1.2501\sqrt{B_{lw}}$, where B_lw - Length of the boat at waterline.
- Force of wind load: $F_w = \frac{1}{2}DA_s^2S_a$, where D - Density of the air, S_a - Area of the sail.
- Boat acceleration: $B_a = (F_w + F_d)/B_m$, where F_d - Drag Force, B_m - Boat mass.

Having conducted such calculations one could have a theoretical solution to some of the crucial problems of design and implementation of the sailing boat.

Nevertheless, it was decided not to move forward with such attitude of calculations. Because expected result needs to be much precise than could be received by such calculations. When boat moves through different obstacles the estimations and further calculations should lean on the safest possible variants. So, when the environment could not satisfy stable true wind such solution is not likely to give a satisfactory result.

Another option was suggested to disregard the calculations of the apparent wind speed and to focus on sensor measurment filtering to estimate better acceleration, speed, position, and global angle of the boat. Finally, the decision was made not to calculate estimations for the boat by its default parameters and certain environment, but to analyze the behavior and based on it attempt to estimate its boat's future performance.

2.4.4 Rotation Expert

The rotation expert is used to estimate the current rotation of the sailing robot. By kalman-filtering the magnetometer, accelerometer and gyroscope data from the IMU, the current rotation of the three boats axes can be estimated. This information can then be used to estimate the current heading, speed and after all the position of the sailing robot.

Being influenced by different forces the boat becomes a complicated source to receive valuable information from and the entity to control. At this part a precise look at position of the boat taken, namely, Z (yaw) axis according to North, X (pitch) and Y(roll) axis, as shown in Figure 6 from ? (?). For the

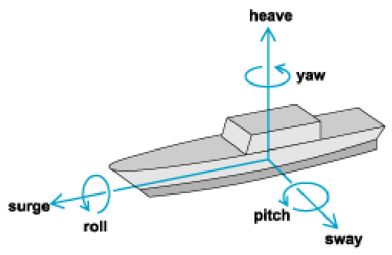


Fig. 6. axes of the sailing boat

position calculations magnetometer, accelerometer and gyroscope data is used. As the data received is noisy sensor fusion should be applied. The team decided to move forward with Kalman filter which was believed to tackle problems with the noise and provide with cleaner data. The jaw calculation with Kalman filter(Figure 7 by ? (?)) is the most important in this section. This could be calculated: $yaw = \arctan(magY \cos(roll) + magZ \sin(roll))/(magX \cos(pitch) + magY \sin(pitch) \sin(roll) + magZsin(pitch)cos(roll))$. The idea behind compensating for the yaw is that having to first rotate the device to a horizontal plane and then find the yaw on that new orientation. It should provide the angle to the North with which the boat moves:

- yaw is between 0 degrees and 67.5 degrees is North-East
- yaw is between 67.5 degrees and 112.5 degrees is East
- yaw is between 112.5 degrees and 157.5 degrees is South-East
- yaw is between 157.5 degrees and 202.5 degrees is South
- yaw is between 202.5 degrees and 247.5 degrees is South-West
- yaw is between 247.5 degrees and 292.5 degrees is West
- yaw is between 292.5 degrees and 337.25 degrees is North-West

The initial angle is assumed to be zero by default. Simple way: With a raw data from one of the sensors noisy predictions are calculated. So, in the default time stamp noisy predictions are being received which are most likely not useful for rotation of the boat.

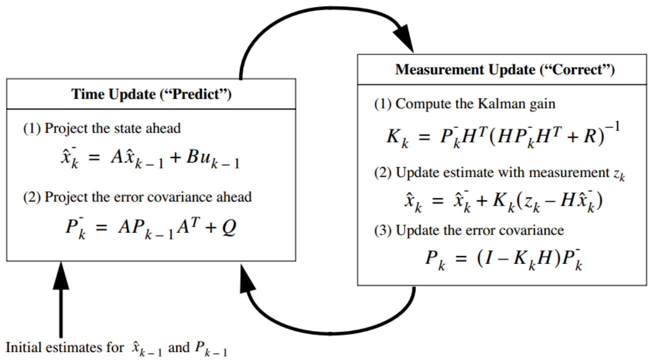

Time Update ("Predict")
(1) Project the state ahead
$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1}$
(2) Project the error covariance ahead
$P_k^- = AP_{k-1}A^T + Q$
Measurement Update ("Correct")
(1) Compute the Kalman gain
$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1}$
(2) Update estimate with measurement z_k
$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-)$
(3) Update the error covariance
$P_k = (I - K_k H)P_k^-$
Initial estimates for $\hat{x}_{k-1}$ and P_{k-1}

Fig. 7. prediction & update step of the used kalman filter

Kalman way (decided way): Magnetometer value from the X and Y are being used to convert them into Z angle of a compass heading: $Z = \arctan(Y/X)180/\Pi$. Then the raw predictions are based on fusing information converted by above mentioned formula and gyroscope Z (yaw) angle for a certain time stamp in Kalman filter. In this filter are three noises counted: process noise, noise gyro bias variance and the measurement noise. Including this factors one could get better estimations than using only one sensor data. So, it results in increasing the likelihood of the successful boat navigation Figure 8. The implementational principles were used from ? (?). The same attitude is used for Y(roll) and X(pitch) angles. The only difference is that the initial data is used by calculating $Y = ACC_Y/\sqrt{ACC_X^2 + ACC_Z^2}180/\pi$ and $X = ACC_X/\sqrt{ACC_Y^2 + ACC_Z^2}180/\pi$. There is also an option to go on gyroscope fusion with accelerometer, but this way was advised by ? (?) not to consider in case using MPU6050. The reason is than innitial readings of a gyro are at most cases much noisy than accelerometer readings. Considering this, advide it was decided to err on the side of fusion explained above.

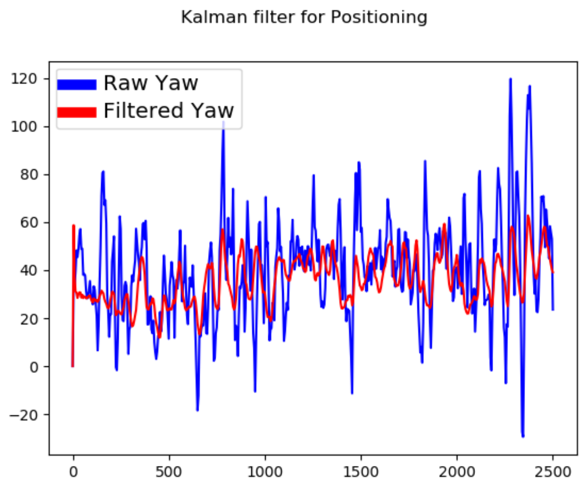


Fig. 8. graph of filtered data by the kalman filter

The used Kalman filter minimizes some noise and makes the rotation calculations more precise. Therefore it increases the chances of success of the navigation system by providing more detailed information about the sailing robots current state.

2.4.5 Position Expert

The position expert was intended to estimate the current position of the sailing robot. The idea was it to perform a sensor fusion of the accelerometer and the gps sensor. By using a kalman-filter it would also be possible to estimate the future position of the sailing robot depending on the acceleration forces applying to the boat.

By integrating the acceleration of the x- and y- axes over a fixed time interval, the velocity v of the sailing robot can be calculated as follows:

$v(t) = v(0) + \sum a \cdot \delta t$

where $v(0)$ is the velocity at time $t = 0$ and $\sum a$ are all sensor readings performed by the accelerometer during period δt. The formula can be applied for each axis individually.

Outgoing from the position of the sailing robot at time t, the position at time $t+1$ can be calculated as follows:

$pos(t+1) = pos(t) + v(t) \cdot \delta t$

where $pos(t)$ is the current x- or y- position of the sailing robot.

By modeling an appropriate kalman-filter which uses the above formulas for its prediction step and the gps sensor reading for its correction step one could get a better position estimation than just using the gps reading. Because the idea for this position expert raised in the late phases of the project where several other tasks had to be finished, and Ostap already focused on estimating the position as described in the rotation expert section, the idea was discarded and the kalman-filter not implemented.

2.5 Simulation

2.5.1 Comparison of simulation environments

As the aim of the project was to develop an authonomous boat and the costs of its testing in real water environment are large, the team decided to use simulation environment. Simulation is the tool to reproduce the behavior of a system, which can be used to gain new insights about that system and estimate the performance without using time and cost consuming real environment tests.

Our team has encountered a simulation project ? (?) that has a German language graphical interface and simulates how an engine boat moves under certain physical conditions from the starting point to the port. This conditions could be set manually so it allows to test the implemented sailing algorithm. Physical condition variables are: float velocity, float angle, wind velocity, wind angle. The boat can be placed at the certain point and with a certain angle to the port. Speed of the boat could also be applied. Display shows the actual angle or boat move, the correction angle and the angle of a wind. Although this simulation looks competitive, the decision was taken not to move forward with it. The main reason is that it was designed for simulation of engine boats, and there were no documentation or guidance.

After careful investigation of the topic three simulation environments were selected for further exploration: V-Rep from Coppellia, Gazebo and MOOS-ivp from Oceanai. In the comparison of V-Rep and Gazebo the article of ? (?) was used. Their main characteristics are described in the following table.

Simulation environment	Advantages	Disadvantages

V-Rep from Coppellia	- advanced 3D graphics - cross platform and portable - ODE, Bullet and Vortex physics engines can be used - 7 fully supported programming languages (including Java). The default language is LUA. - more than 100 Remote API functions - opportunity to simulate sensors as well as generate sensor data is included - collision detection, path/motion planning and dictance calculation modules are included - data recording and visualisation is included - a set of already made models as well as the opportunity of importing external models is included - water surface model is included	- the water surface model has no physical features. The Archimed law as well as wave movements have to be implemented with direct force adding - there is no boat 3D model already included - work with imported 3D model is time consuming: the appropriate model should be chosen, its physical parameters should be added. Imported model also requires proper division into parts and accurate connecting of them
Gazebo	- advanced 3D graphics - cross-platform portable - ODE, Bullet, Simbody and Dart physics engines can be used - the opportunity to run the simulation in online server is included - the opportunity to simulate sensors as well as generate sensor data (optionally with noise) is included - the set of already made models as well as the opportunity of importing external models are included	- there is no boat 3D model already included - work with imported 3D model is time consuming: the appropriate model should be chosen, its physical parameters should be added. Imported model also requires proper division into parts and accurate connecting of them. - no water surface model included

MOOS-ivp from Oceanai	- very simple 3D graphics, which allows to concentrate more on model‘s behavior and mission planning - cross platform - developed specially to support operations with autonomous marine vehicles as well as simulations with them - a set of basic predefined missions and vehicle behaviors is included	- support of the IvP Helm and related behaviors on Windows is limited - allows only C++ modules to be involved

Considering the facts that V-Rep simulator has collision detection and path planning modules as well as that it supports Java, which is the main programming language of the project, V-Rep simulation enviroment was chosen for creating simulation at first. After estimating the time needed to develop accurate 3D model and water environment in V-Rep, MOOS-ivp with simpler graphics and orientation on specifically marine missions was chosen.

2.5.2 Additional functionality of chosen simulation environment

There also appeared another reason to choose MOOS: MOOS-IvP consists not only of instruments for simulation purposes, but also of modules that can be used as way of controlling real unmanned vehicle. MOOS provides Middleware capability by using publish-subscribe architecture. The processes communicate with each other by using MOOSDB (Mission Oriented Operating Suite—Database). The backseat driver algorithm is used in the autonomy architecture of MOOS. The autonomy system is located in a distinct computer, while the control system locates on a vehicle computer. The distinct computer provides vehicle a set of autonomy variables, such as desired heading, speed etc. The vehicle computer sends back the information about vehicle position, trajectory etc. as it is shown in Figure 9.

The explained approach can be used in the future work with a project as an alternate version of real vehicle autonomy architecture. More information about that can be found in the article of ? (?).

2.5.3 Simulation implementation with moos-ivp

The goal of the simulation task was to develop a simulation of the boat travelling

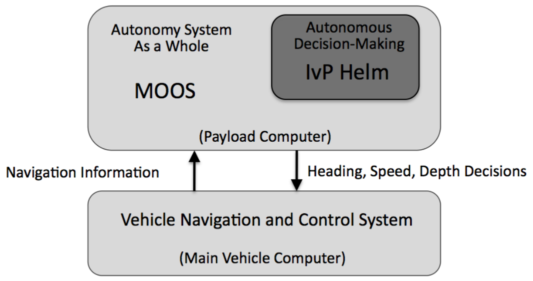

Autonomy System As a Whole
MOOS
Autonomous Decision-Making
IvP Helm
(Payload Computer)
Navigation Information
Heading, Speed, Depth Decisions
Vehicle Navigation and Control System
(Main Vehicle Computer)

Fig. 9. backseat driver algorithm

from the starting point to destination. During the simulation run the algorithm that defines heading for real vehicle should receive the simulated boat‘s current position. The simulated boat should change heading direction in regard to continuous changes of the heading in algorithm output.

For the implementation of simulation two repositories were used: moos-ivp and moos-ivp-extend. The latter one is a set of already predefined C++ modules for providing autonomy of robotic platforms. The former is a tree for building one‘s own extentions to MOOS-IvP.

MOOS-IvP simulations are behavior based. That is why the implementation mainly depends on two types of files: mission files and behavior modules. Mission files are .moos file with mission configuration and .bhv file with configurations of all the behaviors used for specific mission. The behavior modules can be found in source libraries and are C++ files, each of which describes a specific behavior.

For our simulation `BHV_HeadingChange` from `lib_behaviors-marine` was chosen as a starting point. It was then rewritten regarding the needs of our project. Such variables, as current GPS coordinates, current heading, heading goal and heading delta were chosen to control the simulated boat. Two datagram sockets were added. One socket is used to pass from Middleware to simulation the needed heading of the boat in the form 'rudder(heading_delta)'. Another one socket is used to pass from simulation current decimal GPS coordinates, current heading of the boat and timestamp in the form 'gps_dec(timestamp, current latitude, current longitude, current heading)'. Due to these changes the boat can continuosly change its direction in response to new path algorithm output during the simulation run.

2.6 Testing

During the development process of the Middleware the following software parts have been extensively tested to guarantee the correct functionality of the software:

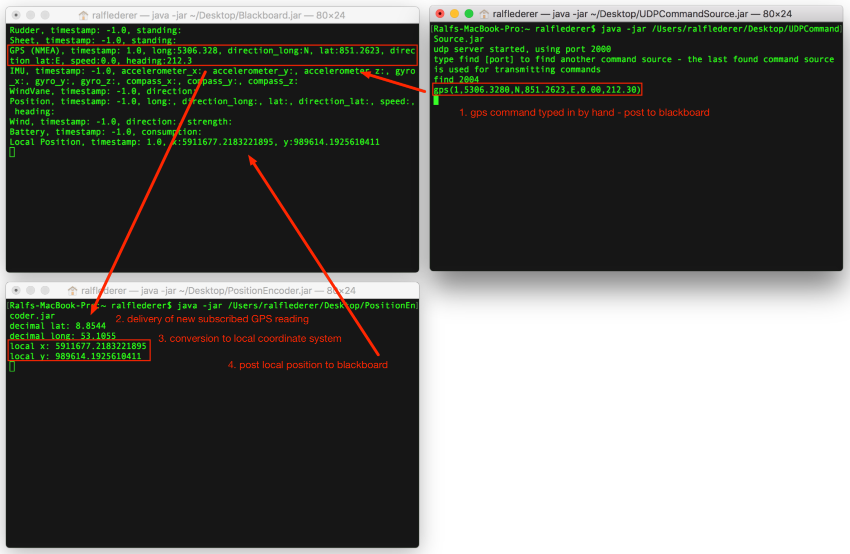
1. The USART interface on different platforms using an ATXMega sending dummy commands
2. UDP transmission between command sources and experts
3. The subscription functionality of experts
4. Sensor implementations regarding to command processing
5. Communication of standalone instances of experts distributed over a network
6. Parsing of control commands
7. Correct functionality of tasks executed by individual experts

To ensure the correct data distribution within the Middleware network of experts a terminal application representing a simple command source was created. This application is able to send control commands typed in by hand to any other command source or expert. Further more standalone instances of all experts have been created printing out the current state of their attached sensors to the console. By sending control commands one is able to evaluate the correct processing of the commands by observing the console outputs of the addressed experts. Further more the subscription process can be observed and evaluated. Points 1-5 have been manually evaluated using this technique. An example of an evaluation is visualized in the Figure no. 10. To test the correct parsing of commands (6.) a unit test was created in which some example commands were parsed and observed if the matching of the regular expressions are correct. The tasks executed by specific experts were evaluated dynamically. For the local position encoder e.g. a set of test values from the gps sensor was taken and each conversion step was calculated by hand to check if the outputs of the expert are correct.

Execution Criteria

The transmitting of data between nodes and especially the blackboard is decisive for the system to operate correctly. To ensure the desired functionality of the system, the following criteria have been specified:

1. A system which hosts Middleware nodes, must provide transmission interfaces which are fast enough to process the communication between nodes and between the hardware and the middleware.
2. The maximum message load of transmitted messages must be at least as big as the amount every possible command, specified in the used communication protocol, produces. This ensures that all control commands are able to be sent and processed correctly.
3. A system running Middleware nodes must provide enough cpu power to ensure their correct execution.


ralflederer — java -jar ~/Desktop/Blackboard.jar — 80×24
ralflederer — java -jar ~/Desktop/UDPCommandSource.jar — 80×24
1. gps command typed in by hand - post to blackboard
ralflederer — java -jar ~/Desktop/PositionEncoder.jar — 80×24
2. delivery of new subscribed GPS reading
3. conversion to local coordinate system
4. post local position to blackboard

```
Rudder, timestamp: -1.0, standing:
Sheet, timestamp: -1.0, standing:
GPS (NMEA), timestamp: 1.0, long:5306.328, direction_long:N, lat:851.2623, direc
tion_lat:E, speed:0.0, heading:212.3
IMU, timestamp: -1.0, accelerometer_x:, accelerometer_y:, accelerometer_z:, gyro
_x:, gyro_y:, gyro_z:, compass_x:, compass_y:, compass_z:
WindVane, timestamp: -1.0, direction:
Position, timestamp: -1.0, long:, direction_long:, lat:, direction_lat:, speed:,
 heading:
Wind, timestamp: -1.0, direction:, strength:
Battery, timestamp: -1.0, consumption:
Local Position, timestamp: 1.0, x:5911677.2183221895, y:989614.1925610411
```

```
Ralfs-MacBook-Pro:~ ralflederer$ java -jar /Users/ralflederer/Desktop/UDPCommand
Source.jar
udp server started, using port 2000
type find [port] to find another command source - the last found command source
is used for transmitting commands
find 2004
gps(1,5306.3280,N,851.2623,E,0.00,212.30)
```

```
Ralfs-MacBook-Pro:~ ralflederer$ java -jar /Users/ralflederer/Desktop/PositionEn
coder.jar
decimal lat: 8.8544
decimal long: 53.1055
local x: 5911677.2183221895
local y: 989614.1925610411
```

Fig. 10. manual evaluation of gps command processing with blackboard and local position encoder

1. speed of message passing

The speed of message passing between nodes mainly depends on the network infrastructure as well as the message overhead. Since nodes can be hosted on different machines the connection between the machines is crucial. When all nodes are hosted on the same machine on the local host the speed is only limited by the machines hardware. Packages exchanged over the local host are not processed by the network interface controller (NIC) of the machine. They are processed by a loop back adapter of the kernel which enables high message passing speed. Otherwise the network interface speed between systems running Middleware nodes must be chosen according to the amount of messages exchanged between these machines.

The communication between the boats hardware and the Middleware is provided by a serial interface. The specified baudrate of the interface limits the speed as well as the data throughput. It needs to be chosen according to the network traffic produced by sent control commands. One can say that the communication between the boats hardware and the Middleware is the bottleneck of the system. If the number of sensors attached to the boat increases in the future or in other projects, the communication needs to be reorganized accordingly.

2. maximum message load

The message size for messages has been limited to 2MB by software. This is sufficient for the used communication protocol. If messages with larger sizes than 2MB are received, only 2MB of the data is processed. Further more a message will be raised, indicating that an invalid command was received. If the infrastructure is used in other projects which have a higher message load the size can be adapted accordingly. Technically the size is limited to the maximum transmission unit (MTU) of the used network.

3. cpu load of hosting nodes

Because the Middleware is written in JAVA, software processes hosting nodes need to run the java virtual machine. This produces basic cpu load. Further more the amount of messages sent and received influence the cpu load needed. The most load is created by the specific task a node is executing. For example a node executing a complex kalman filter will use more cpu power than a node which just executes a conversion between coordinate systems. To calculate the cpu power needed by the whole Middleware network, the consumption of each node has to be inspected individually.

2.7 Conclusion

The middleware provides a suitable software layer for abstracting the sailing robots hardware and it enables data access and controlling to higher level tasks, like the navigation. It is designed as a network of nodes, namely experts, each responsible for a specific tasks. For the sailing robot project experts for sensor fusion, filtering and position converting have been implemented. Unfortunately not all planed nodes could have been implemented in the in the available time. Because of the shared architecture of the Middleware it was able to add a simulator node for simulating the sailing robots behavior under real world conditions. This was a great benefit since testing in the real environment was not possible. Linking together the Middleware with the hardware and the navigation layer figured out to be tricky in the beginning. In the end, however, it worked successfully and all of the three software layers were able to communicate using the provided Middleware. By abstracting the implemented communication protocol from explicit transmission techniques, more and other transmission techniques can be added in the future. The communication protocol, specifying the interface between the used hardware and the software, can easily be extended to handle other than the implemented control commands. This makes the provided Middleware able to be used in other projects as well. It is applicable in systems that need to abstract from specific hardware to perform tasks based on hardware-collected data and to control hardware actuators, e.g. servo motors by sending control commands.

3 Navigation

The navigation is a high-level subsystem that offers functionality used to calculate a route for the boat. Navigation tasks and resulting plans are sent and received using the middleware.

3.1 Introduction

We have chosen to focus on finding a *safe* path that reaches a predetermined goal. A *safe* path is considered to be navigable by the boat. We came up with an approach that aims to respect the following properties of the sailing boat:

- Sailing boat is able so safely navigate to waypoints and goals proposed by the navigation system.
- Both right of way rules and additional rules imposed by the team are respected.
- In an unexpected failure, the boat falls back to an emergency protocol.
- In case of an unexpected change of position, a new route should be calculated to sustain a safe path.

To sum up, our research was aimed to answer following issue:
How do we design a navigation algorithm that enables a boat to autonomously sail from its position to a predefined goal while taking obstacles into account?

The environment of the sailing robot as observed by sensors on boat and on shore contains a lot of known, partially known and unknown variables. Seemingly insignificant factors may lead to significant changes of the state of the boat. The impact and the accuracy determines the relevance of a factor. For example, small waves can change the direction of the sailing boat (also known as heading) quite a lot, but sensors only output vague results. To abstract from these issues, we assume the sensor data provided to the Navigation to be valid.

Following factors need to be taken into account:

Position of the boat and goal is expected it to be very accurate. The goal position needs to be defined a priori.

Terrain includes the geographic boundaries of the environment such as coastlines and shallows that are not navigable. A nautical map is needed in order to model navigable waters.

Parameters of the Ship describe the behavior of the ship based on environmental factors like wind direction and speed. The efficiency of the boat with respect to the direction the wind is coming from is especially important. These parameters need to be observed experimentally, so changes to the algorithm need to be possible.

We decided to use a planning algorithm in a rasterised representation of the map because it allows for a well defined model of the environment: Distances between cells are proportional to the dimensions of the grid:
$distance(A, B) = distance(A + C, B + C)$. Also, distances are commutative $distance(A, B) = distance(B, A)$. Additionally, directions can be calculated only using coordinates: $direction(A, B)$. A,B and C are coordinates in the rasterized map.
We see the advantages in these properties of the approach over others, e. g., a vector based approach proposed by Stelzer (?, ?).

In contrast to a motor boat, the wind keeps the sailing boat from going directly to its destination, so it needs to be considered because it determines the efficiency of the boat. An example of a polar diagram showing the efficiency can be seen in figure 15. There is a range of angles also known as the *no-go zone* a boat is unable to manoeuvre.

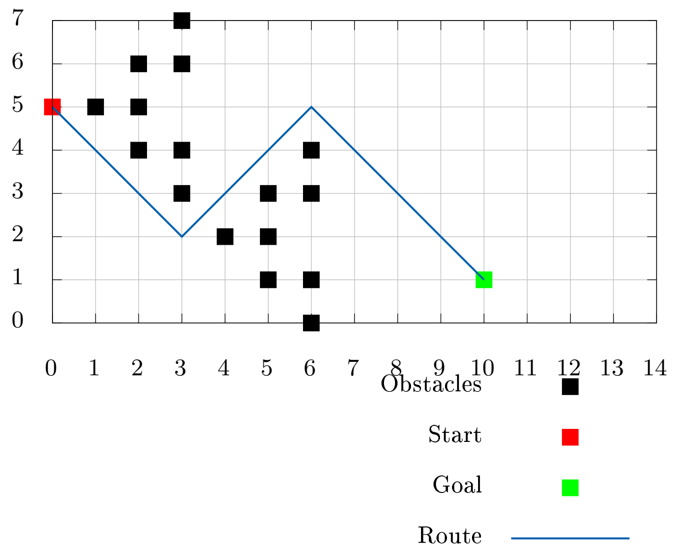


Fig. 11. Example of the approach using moving obstacles as a model for *no-go zones* of the sailing boat. This example was manually created.

We defined a moving obstacle that models this *no-go zone* positioned relative to the boat. This *wall* consists of non-navigable cells which restricts the boat from navigating against the wind. The *wall* is defined by the range the boat is unable to navigate to, its position relative to the boat and its length. A toy example can be seen in figure 11. Two walls block a range of 90° relative to the boat.

We decided that the method was unsuitable because of a number of reasons. Too many questions regarding implementation remained unanswered:

Characteristics of the wall were very hard to determine. How wide must the wall be? Does the width of the wall change? How big is the *no-go zone*?

Moving the wall implies modifying the cost of cell transitions, which is intended in D*. How often and when should the wall move?

Narrow passages render a problem for this approach. While a narrow passage such as a channel (figure 22) can be navigable in headwinds, the wall may block it completely. How can we avoid blocking narrow, but navigable passages such as channels?

The approach that is proposed in the following overcomes the described problems by introducing a tree dimensional configuration space. The state of the boat consists of the two coordinates and additionally, the heading. The approach respects the true wind angle in the cost of state transitions. We can additionally impose the cost of turning the boat directly since this requires a state change.

3.2 Implementation

3.2.1 Task Representation

We model the environment of the sailing boat as a rectangular grid consisting of square cells. Each cell is either navigable (water with enough clearance) or not, such as land, shallow water or other boats. We define a map $M = (dim_x, dim_y, O)$ as a $dim_x \times dim_y$ grid where O is the set of obstacles i.e. non-navigable cells. Representation of the earth's surface into a two dimensional grid introduces negligible errors. However, in a representation with square cells, calculations are easier to perform because of equal distances between neighbors.

Wind direction is stored as the angle in degrees from which it originates, starting at $0°$meaning northerly wind turning clockwise.

In order to model the position of the boat relative to the wind, we introduced the heading as the third dimension of our configuration space. The heading of the bow is also an angle analog to the wind. We define a start as the initial configuration of the boat and a goal as the desired position with an arbitrary heading.

We define the configuration of possible boat positions as $C = (x, y, \theta)$ consisting of its position and heading. A task $T = (M, C_{start}, C_{goal}, C_{boat}, \phi)$ is specified by a map M, the configurations of the start, the goal and the boat and the global wind direction ϕ.
Figure 12 shows an example with 4 possible headings.

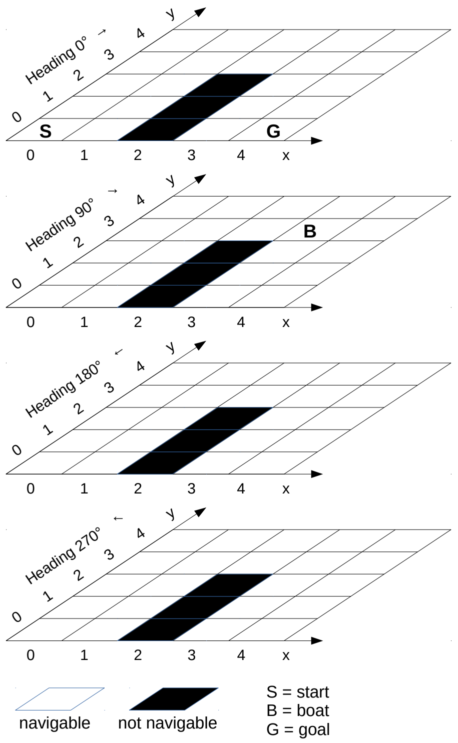


Fig. 12. Task $T_{example}$

The task can be described as the following:
$T_{example} = (M_{example}, C_{start}, C_{goal}, C_{boat}, 40°)$

- $M_{example} = (5, 5, \{(2, 0), (2, 1), (2, 2)\})$
- $C_{boat} = (3, 3, 90°)$
- $C_{start} = (0, 0, 0°)$
- $C_{goal} = (4, 0, 0°)$

3.2.2 D* Algorithm

Search Space

We assume that the boat may move to one of its eight neighbors not changing the heading or turn in any other direction while not changing its position. Therefore, every state has a maximum of $8+(|possible\_headings|-1)$ neighbors depending on its position. We assume that a change of position and direction are not possible at the same time because the efficiency of the boat performing such a complex move is hard to observe. The maximum number of edges connected to one state would also increase to $9 * |possible\_headings| - 1$.

The direction and distance to each of the eight neighbors in two dimensions can be seen in figure 13.

Δy \ Δx	-1	0	1
1	315° 1.414	0° 1	45° 1.414
0	270° 1	current position	90° 1
-1	225° 1.414	180° 1	135° 1.414

Fig. 13. directions in degrees and distances to neighbors in 2D

Cost of turning the boat

Having a third dimension in the configuration enables us to propagate the loss of speed due to turning in the state transition function. An optimal path contains as little turns as possible. Figure 14 abstracted from the sailing environment shows the path calculated for the task $T_{example}$, which includes turns.
In the example, a path from $C_{start} = (0, 0, 0°)$ to $C_{goal} = (4, 0, 0°)$ was calculated. The four different layers represent the headings of the boat $\{0°, 90°, 180°, 270°\}$. The boat starts in layer 0°, from which it can change its heading or move in any direction. The cost of moving to its neighbor $C_{10} = (1, 0, 0°)$ is $infinite$, since it is not in the indicated direction. Instead, the move from C_{start} to $C_{01} = (0, 1, 0°)$ was chosen.

Algorithm description

We used the planning algorithm D* (dynamic A*) proposed by Anthony Stentz (?, ?) to find the cheapest path from a start to a goal configuration of a task. In contrast to A*, it starts at the goal configuration looking for the cheapest path to the start thereby reversing the task.
We determined that there are two major advantages over using A* with respect to motion planning:

Change in Position The boat will not always follow the path calculated by the algorithm. In the event of such an unexpected change in position, the path does not need to be recalculated completely. In most cases, a path to the new position of the boat has been already computed.

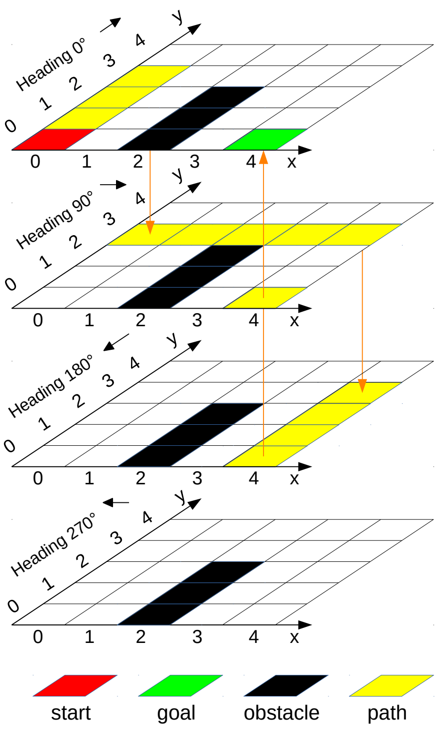


Fig. 14. Path calculated for $T_{example}$ (without wind)

Change in Environment Changes in the environment at runtime are also common and need to be taken into consideration. Other boats changing their position or parts of the map no longer being navigable are some of these changes. Modifying the costs of a state transition is possible and intended in D*. Recalculating the route with new costs only requires a local computation.

Like in A*, all states go through a life cycle, initially they are NEW states. After the first visit, they are added to the queue of OPEN states. Finally, when no shorter path is possible, they are CLOSED. Note that not all states have to be closed for the algorithm to terminate and CLOSED states may be OPEN at a later point in time when costs are modified. A priority queue maintains all states that are OPEN, sorted by the minimum estimated cost to the start configuration. Similar to stored paths in A*, every state has a *back pointer* that points to the next state in the shortest path to the goal. A path can be constructed using these pointers.
The algorithm mainly uses two functions: *process-state* and *modify-cost*. The function *process-state* processes the head in the queue of OPEN states and is invoked iteratively until a path is found. This is the case when the start state is CLOSED or it is determined that no path exists. If the cost between two configurations change, *modify-cost* updates these changes in affected states.

process-state The head of the queue of OPEN states X is removed. Every neighbor Y of X is visited, thus paths with lower costs can be found. NEW states in Y are added to the open list (queue) and receive an initial path cost value and back pointer to X. When Y is OPEN already, existing paths may be redirected if a shorter path is found.

modify-cost The arc cost function from X to Y is updated. State X is inserted in the OPEN list. Further calculations and propagations are done using *process-state*.

Polar Diagram

The speed of the sailing boat relative to the wind speed depends on the direction the wind is coming from. The efficiency varies from boat to boat, so the values of the polar diagram need to be tuned empirically for each boat. For our implementation, the values were retrieved from a website with data for real boats[1]. These values can be examined in figure 15.

Calculating the Cost

We define the cost function between two states as a fraction dividing the distance by the efficiency of the boat with respect to the wind as stated in figure 16. This cost function abstracts from the grid representation of the map as it only considers the position of the boat and assumes the boat is heading to B. The

[1] https://www.seapilot.com/features/polars/download-polar-files/

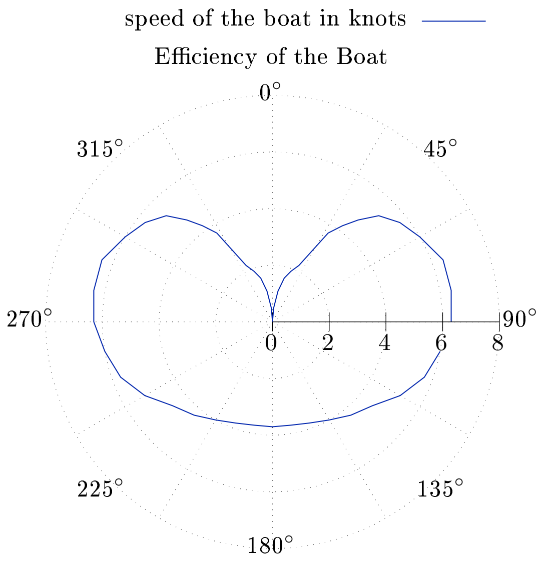


Fig. 15. Polar Diagram indicating the efficiency of a sailing boat at $windspeed = 6knots$

function $polar(direction, windDirection)$ returns the efficiency of the boat based on the heading of the boat relative to the wind, in essence, it looks up the value for $((direction - winddirection) \mod 360)°$ in the previously defined table.

$$cost(A, B, wind) = \begin{cases} \frac{|A-B|}{polar(dir(A,B),wind)} & navigable(\overline{AB}) \\ infinite & otherwise \end{cases}$$

Fig. 16. Cost from A to B with wind

Changing the position in a two dimensional grid to a neighboring cell only allows for eight different directions. Intermediate steps are not possible and intuitively, the boat direction of a state may only have eight values as shown in figure 13. In order to navigate in more than just eight directions, we no longer restrict the heading of boat configuration to multiples of 45°. With the boat being able to go to every neighbor with any heading, there exists a difference between the direction $dir(C_x) = \theta$ of the boat configuration and the direction of the movement in the grid, which is necessarily a multiple of 45°. This introduces an error $(\Delta x, \Delta y)$ which needs to be accumulated in each iteration. If the error gets larger than half of the width of a cell, we suggest to block further cells in that direction in order to reset the error. The actual boat position is the center of its cell offset by the error $(\Delta x, \Delta y)$.

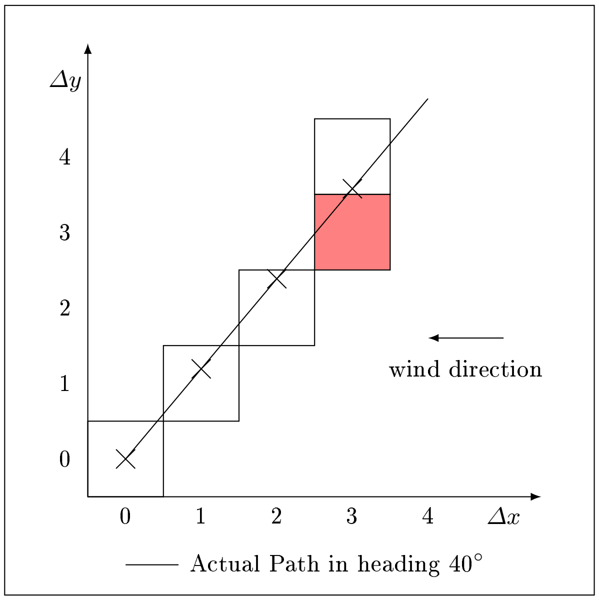


Fig. 17. Actual path vs path in grid, heading 40° is cheaper than heading 50°

With easterly wind, for example, a move from $C_{from} = (0, 0, 40°)$ to $C_{to} = (1, 1, 40°)$ is cheaper than the same move but heading 45° because of higher efficiency with respect to the wind. The absolute offset of Δy increases by $tan(90° - 40°) - 1$ every step. After 3 steps, $\Delta y = 3 * (tan(40°) - 1) = 0.5753$ which is greater than 0.5, so the actual corresponding position of the boat is $(3, 3 + 0.5753) \approx (3, 4)$ instead of $(3, 3)$. Thus, the cell $(3, 3)$ needs to be blocked to reduce the offset as seen in figure 17.

$$courseDiv(C_a, C_b, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} * \exp -\frac{1}{2}\left(\frac{|dir(C_a) - direction(pos(C_a), pos(C_b))|}{\sigma}\right)^2$$

Fig. 18. Weight for the error π introduced by the difference between the movement of the boat α and its movement in the grid.

To prevent the boat from navigating to cells it is not heading to but to allow small deviations like described above, we introduce a weight factor. A smaller weight factor indicates a larger deviation of the movement to the heading. The function $courseDiv(C_a, C_b, \sigma)$ seen in figure 18 is based on the Gaussian normal distribution and is empirically tuned with σ.
The resulting cost function for a change in the configuration from C_{from} to C_{to} can be seen in figure 19. If the heading of the boat changes, we define the cost of the maneuver to be equal to the absolute rotation. Otherwise, we calculate the cost by dividing the euclidean distance by the product of previously calculated

$$cost(C_f, C_t, wind, \sigma) =$$
$$\begin{cases} |dir(C_f) - dir(C_t)| & pos(C_f) \equiv pos(C_t) \wedge navigable(C^\circ) \\ \frac{|C_f - C_t|}{courseDiv(C_f, C_t, \sigma) * polar(dir(C^\circ), wind)} & dir(C_f) \equiv dir(C_t) \wedge navigable(C^\circ) \\ infinite & otherwise \end{cases}$$

Fig. 19. Cost from C_{from} to C_{to} with wind

weight of the error and the efficiency of the boat at the specific true wind angle returned by the function $polar(direction, windDirection)$. Both C_{from} and C_{to} must be navigable denoted by $navigable(C^\circ)$. Otherwise, we consider the cost to be infinite.

Heuristic

Like A*, D* is also an informed planning algorithm, so heuristics can be used in order to minimize the number of visited states in the map. For that, we set the minimal costs k from any state to the start state in the beginning of the algorithm. Every initial value of k must underestimate the real path costs for the algorithm to return an optimal solution. The heuristic must be admissible. We propose the two heuristics *motor_ boat* and *linear_ distance* which compute initial values for k.
For both heuristics, we use the minimum cost of covering a distance equal to the width of a cell, i.e., the cost under optimal wind conditions, as a distance measure.

linear_distance Returns the euclidean distance from each state to the start state. If the state is not navigable, it will return $infinite$. The heuristic is admissible.

motor_boat Returns the cost of piloting a motor boat from each state to the start state. The cost of covering a distance x with the motor boat are set to the minimal cost of covering the same distance with the sailing boat. In other words, traveling the path under optimal wind conditions. This ensures that the estimated path cost are always less or equal than the real cost. The heuristic underestimates real cost, thus it is admissible.

Properties

Completeness If a path exists and there are finite amount of states, the algorithm will return a solution after a finite number of calls of *process-state*. Otherwise, *process-state* will iteratively visit all accessible states from the goal and in the end, return -1. The start state still remains NEW as it was not visited. Thus, the algorithm is complete as also described in (?, ?).

Optimality Given our cost function, D* returns the path with minimal costs as the used heuristics are admissible.

Complexity D* has a worst-case complexity of $\mathcal{O}(n^2)$ with n being the number of cells (?, ?).

3.2.3 Integration into Middleware

The navigation is a high-level subsystem that can operate independently of the hardware and the Motion Controller. Therefore, the navigation should only be accessible via the Middleware. For this reason, the Middleware provides the blackboard expert. It is the centralized data storage of the application every subsystem has access to and is used to communicate between them. The blackboard manages data in terms of sensors. That means that every information that is stored on the blackboard has to be a sensor reading of a (virtual) sensor.
We identified four virtual sensors that are necessary to implement a simple interface to the navigation:

LocalMap Represents a task $T = (M, C_{start}, C_{goal}, C_{boat}, \phi)$ with $C_{boat} = C_{start}$ that consists of the map M, the configurations of the start and the goal and the global wind direction ϕ. Whenever the task changes, e. g., the goal or the global wind direction has changed, the LocalMap sensor needs to be updated. Then, the navigation solves the task by calculating a path from the start to the goal.

Obstacle Represents the position of an obstacle $o = (x, y)$ that should be added to or removed from the map. An obstacle could be an opponent that moves through the map or cells of the map that are detected to be not navigable anymore. Whenever an obstacle is added or removed, the navigation updates the cost function from and to the obstacle with *modify-cost*.

BoatConfiguration Represents the current boat configuration $C_{boat} = (x, y, \theta)$ of the task defined by *LocalMap*. Whenever the boat configuration is updated, the navigation recalculates the path and stores the next waypoint of the path on the blackboard.

WayPoint Represents the next waypoint $C_{waypoint} = (x, y, \theta)$ to which the boat should navigate.

Since both the obstacles and the boat configuration are part of the task T specified in the *LocalMap*, these three sensors could have been combined. But as already discussed in section 3.3.2, the planning algorithm D* has two special properties with respect to motion planning. First, it can handle a change in the environment, like added are removed obstacles, by locally adjusting the path instead of completely recalculating it. Secondly, if the boat deviates from the path, in most cases, the path does not need to be recalculated. To take advantage of these two properties, the virtual sensors *Obstacle* and *BoatConfiguration* were outsourced from *LocalMap*.

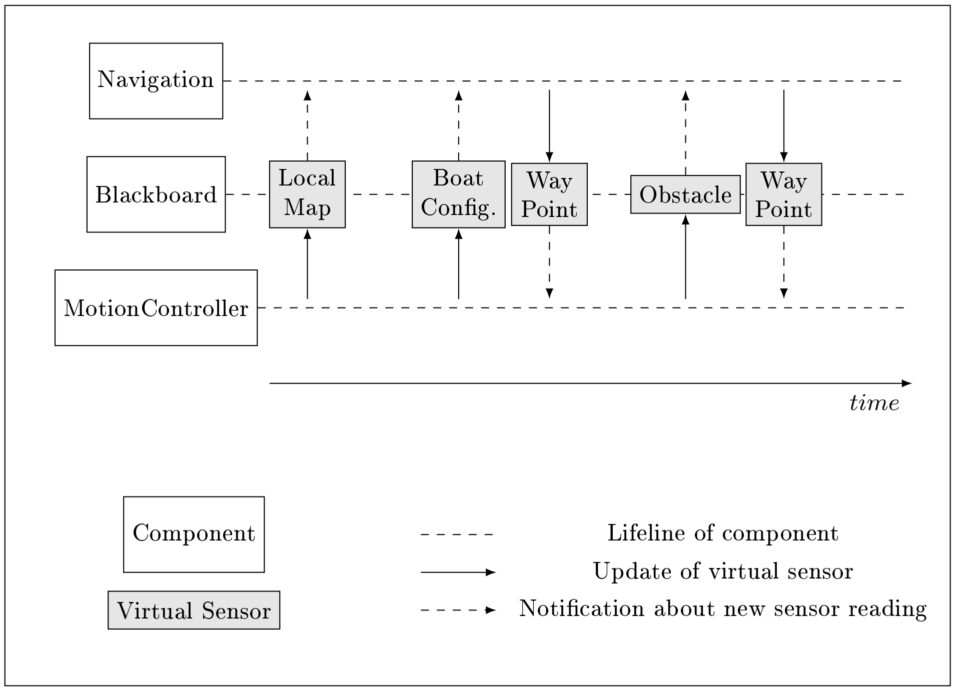


Fig. 20. Example of a sequence of virtual sensor updates for communicating between the MotionController and the Navigation.

Figure 20 shows a typical sequence of virtual sensor updates that are used to communicate between the MotionController and the Navigation:

1. Motion Controller initializes the *LocalMap* sensor on the blackboard with the current map of the environment.
2. Navigation gets notified about the map and calculates a path from the start to the goal.
3. Controller updates the *BoatConfiguration* on the blackboard.
4. Navigation gets notified about the boat configuration and stores the next *WayPoint* on the blackboard.
5. Controller updates an *Obstacle* on the blackboard.
6. Navigation gets notified about the obstacle, recalculates the path and stores an updated *WayPoint* on the blackboard.

3.3 Evaluation

In this section, the heuristics and the runtime of the navigation algorithm are evaluated. Additionally, the practicality is examined and possible improvements are suggested.

The functionalities are tested using manually generated maps. Maps 21 to 27 in the appendix have special features such as good wind conditions or obstacles in the form of islands or narrow channels. Furthermore, maps with no obstacles were created. In these the wind always comes from 90°(E). The boat needs to tack (navigate from W to E), jibe (navigate from E to W) or go diagonally (navigate from SW to NE). The size and the direction are indicated in the name: *100x100-tack* means a map with size 100×100 where the boat has to tack against the wind. Start and goal are at the very edge of the map except for a few maps titled *half* where a boat only has to navigate through half of the map.

3.3.1 Heuristic

Two heuristics are proposed. They aim to improve the performance of the D* algorithm by reducing the number of visited states. More informed heuristics are able to estimate the real cost better and therefore fewer states need to be visited during runtime. This allows the number of closed states to be used to determine how good or bad the heuristic estimates the actual costs.

Figure 28 shows the number of closed states after calculating the shortest path using different maps and the two proposed heuristics. The baseline is the number of closed states without a heuristic. The improvement of the heuristics relative to the baseline is shown in brackets.

map	baseline	motor boat		linear distance	
twoIslands	7201	7194	(−0.1%)	7156	(−0.6%)
twoIslands-short	3841	1827	(−52.4%)	3489	(−9.2%)
goodWindConditions	961	77	(−92.0%)	77	(−92.0%)
vTurn1	1684	844	(−49.9%)	1506	(−10.6%)
vTurn2	1965	1881	(−4.3%)	1746	(−11.1%)
50x50-tack	99239	98455	(−0.8%)	98049	(−1.2%)
50x50-jibe	94481	93807	(−0.7%)	93804	(−0.7%)
50x50-diagonal	98001	96345	(−1.7%)	96998	(−1.0%)
50x50-half-tack	75849	72126	(−4.9%)	72569	(−4.3%)
50x50-half-jibe	70041	67688	(−3.4%)	67634	(−3.4%)
50x50-half-diagonal	68249	66375	(−2.7%)	67062	(−1.7%)

Fig. 21. Number of closed states using different maps and heuristics

The table shows a wide range of improvements from 0.1% to 92%. For the map *twoIslands* and all 50×50 maps both heuristics show only marginal improvements. The number of closed states is reduced by less than 5%. In the map *twoIslands-short* the *motor_boat* heuristic leads to an improvement of more than 50% while the *linear_distance* improves the number of closed states by about 9%. For the map *goodWindConditions* both heuristics lead to the same improvement of 92%.

Both the *motor_boat* and the *linear_distance* heuristic use the maximum speed of the boat in optimal wind conditions to estimate the minimum costs. The only difference is that the *linear_distance* also considers paths via non-navigable cells.
For the map *goodWindConditions*, where the sailing boat sails in only one direction in optimal wind conditions, the *motor_boat* and the *linear_distance* heuristic should be equal to the actual path costs and are therefore very informed. This is the reason why the number of states is reduced to only 77 compared to 961 without a heuristic.
In the map *twoIslands* and all *tack* maps the sailing boat has to tack against the wind. As a consequence, the cost estimations by the *linear_distance* or by the direct path of the *motor_boat* are lower than the actual costs. Therefore, they are not as informed. Nearly all states are still considered and the heuristics lead to only small improvements.
In the *jibe* and *diagonal* maps the boat navigates in one direction (explanation, see 3.5.1) but the wind conditions are not optimal. Therefore, the cost estimation is too low, the heuristic is not informed and the improvements are small.
The only anomalies are to be found in the maps *twoIslands-short*, *vTurn1* and *vTurn2*. In all three maps the path of the motor boat is similar to the path of the sailing boat. But even though the wind conditions are not optimal, the heuristics perform better than in the 50×50 maps.
Probably, a combination of the similar path and the nearby obstacles leads to these results. This thesis is supported by the fact that small changes in the environment, like moving the start by one cell from the map *vTurn1* to *vTurn2*, have a high impact on the improvement (from 49.9% to only 4.3%).

To sum up, the improvement by the heuristics depends on the environment. On open water without obstacles and non-optimal wind conditions both heuristics only lead to small improvements. On maps with obstacles the heuristics can lead to improvements of up to 50%. If the wind conditions are optimal the heuristics help reducing the number of visited states by up to 92%.
The *linear_distance* heuristic is, in most cases, underestimating the actual costs and only leads to improvements of up to 11%. The *motor_boat* heuristic in combination with obstacles and narrow channels can partly improve the number of visited states by more than 50%.

3.3.2 Runtime

Figure 29 shows the runtime of calculating the shortest path from the start to the goal with the D* algorithm using different maps and the two proposed heuristics. The baseline is the runtime of the algorithm without a heuristic. The percentage change of the heuristics relative to the baseline is shown in brackets. Additionally, the time needed to calculate the heuristics is displayed for the *motor_boat* heuristic. Calculating the *linear_distance* heuristic only takes 0.03s for the biggest map (*200x200-tack*). Therefore, this runtime is negligible and not listed below.

map	baseline	motor boat			linear distance	
		heuristic	path			
twoIslands	0.23s	0.03s	0.21s	(−8.7%)	0.07s	(−69.9%)
twoIslands-short	0.11s	0.13s	0.03s	(−72.7%)	0.06s	(−45.5%)
goodWindConditions	0.05s	0.09s	0.00s	(−100.0%)	0.00s	(−100.0%)
vTurn1	0.07s	0.19s	0.05s	(−28.6%)	0.07s	(0.0%)
vTurn2	0.07s	0.12s	0.14s	(+100.0%)	0.07s	(0.0%)
50x50-tack	5.63s	0.59s	3.66s	(−35.0%)	3.63s	(−35.5%)
50x50-jibe	1.67s	0.53s	2.55s	(+52.7%)	2.48s	(+48.5%)
50x50-diagonal	4.01s	0.48s	3.22s	(−19.7%)	3.05s	(−23.9%)
50x50-half-tack	3.14s	0.48s	2.72s	(−13.4%)	2.69s	(−14.3%)
50x50-half-jibe	1.74s	0.48s	2.36s	(+35.6%)	2.39s	(+37.4%)
50x50-half-diagonal	2.52s	0.48s	2.22s	(−11.9%)	2.21s	(−12.3%)
100x100-tack	47.76s	1.94s	41.04s	(−14.1%)	38.55s	(−19.3%)
100x100-jibe	10.78s	1.96s	28.32s	(+162.7%)	29.09s	(+169.9%)
100x100-diagonal	35.24s	1.94s	46.02s	(+30.6%)	39.24s	(+11.4%)
150x150-tack	150.24s	4.35s	204.69s	(+36.2%)	202.08s	(+34.5%)
200x200-tack	440.38s	7.77s	838.31s	(+90.4%)	585.46s	(+32.9%)

Fig. 22. Runtime of D* using different maps and heuristics

Calculation of the path without a heuristic in small maps, like the first five maps that have maximum size of 25×10, is relatively fast. The time ranges from 0.05s for *goodWindConditions* to 0.23s for *twoIslands*. The runtime increases with the size of the map. For the 50×50 maps it takes a maximum of 5.63s and for the 100×100 map a maximum of 47.76s.

We can observe that the runtime with heuristics is decreased for the map *goodWindConditions*. As the number of visited states is decreased by 92% by the heuristics (see figure 28), this is expected. But even though the number of visited states is reduced in every case for every heuristic, calculating the route takes longer for some cases. For instance, for the map *100x100-jibe* the runtime is increased by more than 160% for both heuristics. This is unexpected as the number of visited states is reduced in this case too.

Additionally, for some maps calculating the *motor_boat* heuristic takes longer than calculating a solution to the problem itself. Calculating a path in the map *vTurn1* takes 0.07s, whereas calculating the motor boat heuristic takes 0.19s – more than twice as long.

We propose to use the *linear_distance* heuristic. It is fast to calculate and the runtime is reduced for all maps smaller than 100×100 except for the maps where the boat has to jibe.

In a real-time application, a runtime of 5 seconds should not be too much of a problem as the path only needs to be calculated once in the beginning. But a runtime of 47.76 seconds for the 100×100 map renders the navigation algorithm unusable in real-time. Therefore, the maximum size of the map should be smaller or equal to 50×50.

If the width of a cell on the map is equivalent to 4m in the world, the map still covers an area of 200m by 200m. As soon as the boat reaches the edge of the map, a new map can be created and a new path can be calculated.
If a larger map is required, we suggest using a long-term navigation algorithm in addition to the proposed solution for short-term navigation.

3.4 Discussion

3.4.1 Polar diagram

We did not conduct an experiment to determine the speed of the boat relative to the wind direction, i. e. the polar diagram. Instead, the values were retrieved from a website with empirical data for real boats[2]. But we noticed that the boat did not jibe like a normal sailing boat, although the wind came directly from behind. The speed advantage of the boat for jibing is not large enough to compensate for the longer distance of the maneuver. We have adapted the polar diagram in a test to increase the speed advantage. This lead to to the result that the boat jibed in such a situation.
Consequently, a test of the boat in its real environment would be necessary to create a better model.

3.4.2 Calculated routes

The costs of turning are assumed to be linear in the turning angle. Additionally, the costs of moving to a cell the boat is not directly heading to is modeled using the Gaussian normal distribution. The resulting cost function leads to plausible routes. This can be seen in almost all maps. For example, the route in the map *twoIslands* contains as little turns as possible as it uses the full width of the map.

3.4.3 Edge cases

As we have discussed before, the navigation algorithm leads to plausible routes. However, there are also exceptions that it can not handle.
One example is the map 23. This map represents a narrow channel where one cell of the map stretches over the full width of the channel. Even though navigating to the goal is not possible, our algorithm returned a solution where the boat navigates against the wind.
Therefore, to ensure a navigable route, it is necessary to have a width of at least two cells at each point in the map.

[2] https://www.seapilot.com/features/polars/download-polar-files/

4 Motion Controller

4.1 Introduction

The Motion Controller, from hereon just 'controller', is a group of classes that facilitates communication between navigation and middleware. It decides which class receives what sensor data, dynamically sets the sail and translates the navigation output into movement orders for the boat.

4.2 Decisions

We omitted implementing compensation for drift and adverse weather conditions, such as waves, due to the lack of access to real world testing data. Instead, whenever the boat leaves the expected area set by the Navigation, the controller will ask navigation for a new course based on the current boat coordinates. Our first approach consisted of four parts:

– Main: Registers classes to the blackboard instance
– Sail controller: Sets the sail
– Course correction: Sets the rudder
– Turn controller: Responsible for maneuvers, specifically tacks and jibes

The course correction was intended to adjust the boat heading after maneuvers and keep the boat on course, while the turn controller would take over both sail and rudder control to handle maneuvers. The decision whether a turn is a maneuver was to be made by the main controller based on the relation between the required turn and the current wind direction.

We added a BoatState class to hold variables modified and accessed by more than one controller class. After implementing this architecture, we realized that a self-contained turn controller would not work with our 'act-on-receiving data' approach, as it would require receiving position or compass data to know if the maneuver was finished. To solve this problem, we moved the functionality of the TurnController class into the CourseCorrectionController, abolishing the turn controller as an independent class altogether. The decision whether a turn was a maneuver was also moved to into the course correction, as it would now be the only class directing the rudder. We decided to save the wind direction in the BoatState class, as the CourseCorrectionController needed to know the wind direction to decide whether a maneuver takes place, but otherwise doesn't need a subscription to the wind sensor.

4.3 Implementation

The controller is comprised of four classes:

– MainController
– CourseCorrectionController
– SailPositionController
– BoatState

4.3.1 Startup Upon starting the program, the MainController class creates instances of all controller classes and calls for an instance of the Blackboard class. Following that it will register the controller instances as sensor observers. Once the first position data has been sent to the blackboard, the main controller will ask navigation to generate a plan with the available position data.

4.3.2 Control flow The controller classes are subscribed to the blackboard by the MainController class to receive new position data immediately. The subclasses for sail- and rudder-control only act upon receiving such sensor data. The current boat direction is converted into degrees from compass data received from the boat's sensors, as shown in Figure 30. When the controller has received both a boat position and a waypoint from navigation, it will calculate the difference between the current boat direction and the required heading, from hereon called 'target direction', to move towards the new waypoint. The angle difference between boat direction(A) and target direction(B) is then calculated as in Figure 31. As the boat can turn both starboard and port, angles on the port side are returned as negative numbers down to -180°. The resulting angle difference value is used by the CourseCorrectionController to decide on a rudder angle to turn the boat in the desired direction. Communication between controller and boat happens through virtual sensors that post commands on the blackboard. Whenever the boat reaches a new position, the controller will ask navigation for a new waypoint.

$$boatdirection(x, y) = \begin{cases} (90 - atan(\frac{x}{y})) * \frac{180}{pi} & y > 0 \\ (270 - atan(\frac{x}{y})) * \frac{180}{pi} & y < 0 \\ 0 & y = 0 \wedge x > 0 \\ 180 & y = 0 \wedge x < 0 \end{cases}$$

Fig. 23. calculation of boat direction in degree

$$angle = (A - B) \mod 360$$
$$angledifference(angle) = \begin{cases} angle - 360 & angle > 180 \\ angle + 360 & angle \leq -180 \end{cases}$$

Fig. 24. calculation of angle difference

4.3.3 Course Correction The CourseCorrectionController class is responsible for setting the rudder. It receives compass data and position data from the boat's IMU sensor, local position data from GPS, and target directions from Navigation via the blackboard. Upon receiving new local position data, it will ask navigation for a new waypoint. When the CourseCorrectionController receives a new waypoint from navigation, it will save the target heading as a local variable. Once it receives compass data from the boat, it will calculate the current boat direction, also called heading, and the angle difference between the current boat direction and target direction. The boat's heading is saved in the BoatState class to synchronize with the sail controller. Further, the CourseCorrectionController will check if the required course correction would result in a tack or jibe, as shown in Figure 32, by comparing the boat direction and target direction with the last known wind direction saved in the BoatState class. If the target direction differs from the current boat heading, the rudder is set to an angle between -60° and 60° , appropriate to the difference between boat direction and target direction — A larger difference results in a higher absolute angle. The rudder angle is then sent as an integer to the VirtualRudder virtual sensor, which will post it on the blackboard. Additionally, if a tack or jibe takes place, the "isTurning" variable will be set to true in the BoatState class, telling the SailPositionController to tighten the sail.

$$A = angledifference(boatdirection, targetdirection)$$
$$B = angledifference(winddirection, targetdirection)$$
$$C = angledifference(invertedwinddirection, targetdirection)$$
$$turnCheck(A, B, C) = \begin{cases} turning & A > B > 0 \\ turning & A < B < 0 \\ turning & A > C > 0 \\ turning & A < C < 0 \\ \neg turning & otherwise \end{cases}$$

Fig. 25. turnCheck calculation

4.3.4 Sail Controller The SailPositionController class is responsible for loosening and tightening up the sail to facilitate the optimal velocity given the current boat direction. We defined 'optimal' for the boats velocity as being as close as possible to the sail angle shown in Figure 33 due to lack of real world tests providing us more specific data for the boat model used. The used hardware, with only a single servo for both sails, prohibits us from pulling the sail in a specific direction — port or starboard — or setting the sails independent from each other. As the ship can't sail directly against the wind, we factored in a 'no-go zone' in which the sail position will not be modified. We assumed this zone to be 40° in either direction of the oncoming wind. The sail angle is calculated as shown in Figure 34. Based on the previously mentioned 'optimal' angles, we divided the calculation based on wind direction — sailing against the wind uses a different formula than sailing with the wind. If the boat is currently performing a maneuver — tack or jibe — , the SailPositionController will pull the sail in to 5°, reducing the applied drag required to swing the sail around at the end of the turn.

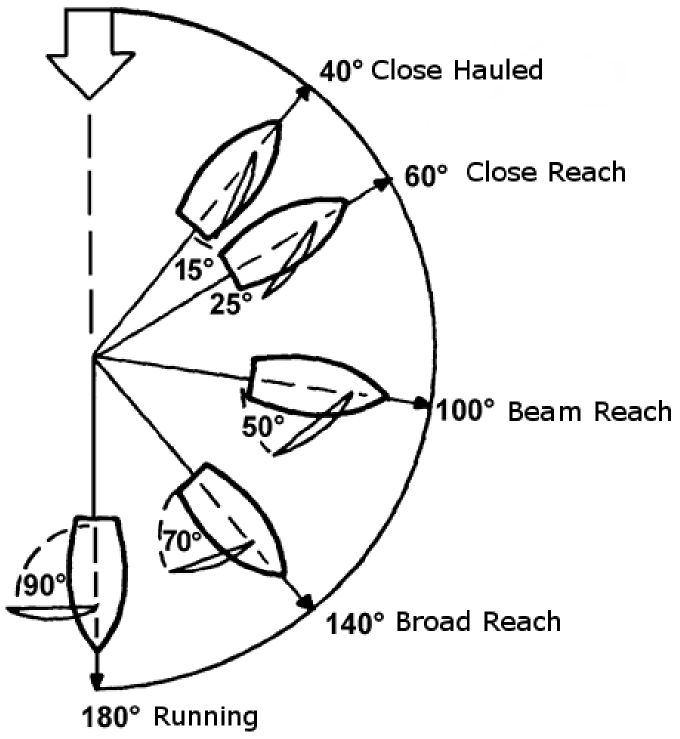


Fig. 26. Sail position based on wind direction

4.4 Testing

The controller was tested by simulating boat sensor input through a linux console. The procedure was as follows:

- run MainThreeDim.java from the IDE
- run MainController.java from the IDE
- run UDPCommandSource.jar from the linux console
- set the blackboard's port in the console with the "find 1337" command
- send dummy sensor data to the board

The sensor data relevant for testing the controller classes are:

$$sailposition(wdir) = \begin{cases} \frac{(wdir-50)}{5*3} + 20 & 50 < wdir \leq 100 \\ \frac{(360-wdir-50)}{5*3} + 20 & 260 \leq wdir < 310 \\ \frac{wdir}{2} & 100 < wdir \leq 180 \\ \frac{360-wdir}{2} & 180 < wdir < 260 \end{cases}$$

Fig. 27. calculation of sail position in degree

– local_position(timestamp, x-coordinate, y-coordinate, heading), the boats position in the local navigation grid, normally calculated in the LocalPosition virtual sensor from GPS
– wind(timestamp, wind direction, wind strength), the current wind direction, normally coming from the Wind virtual sensor
– imu(timestamp, accelerometer_x, accelerometer_y, accelerometer_z, gyro_- - x, gyro_y, gyro_z, compass_x, compass_y, compass_z), coming from the boat's IMU sensor outside of testing. The controller only uses the compass' x and y coordinates and the timestamp, so everything else can be 0.

Our testing parameters were:

– Boat starting position: [0,2], pointing south (180°)
– Wind coming from the east (90°) at strength 2
– The Goal is east at [8,2]
– There are no obstacles on the path

The sail was set to 44° after the first IMU data was emulated manually by console. Upon receiving a target direction of 45° , the controller set the rudder to 40° , and we simulated the boat turning through repeated console input. The system performed as expected with no exceptions. We deposited logs of both the console input and the controller's console output in the resources folder.

4.5 Conclusion

Due to the position of the controller between the navigation and middleware, the development had to react to changes on either side, necessitating adaptive reworks. The controller works well in a theoretical environment, but cannot cope with waves and other environmental effects. Further, we have no direct control over which direction the sail will swing due to hardware limitations, making the boat vulnerable to unstable wind conditions. There are no safe-guards against the boat stalling during a turn maneuver, leaving it unable to navigate.

A drift correction, as originally planned, would require the controller to receive the current boat direction, calculate the required boat direction based on the difference between boat direction and true direction in recent timestamps and steer the boat accordingly, and then send the navigation the direction the boat would have without drift correction to calculate a new waypoint and target direction. This was deemed impractical given the project's time constraints.

5 Conclusion

In the project, we developed a solution to the problem of sailing in partially known environments. It is comprised of the higher-level systems Navigation and Motion Controller which communicate with the boat's hardware via the Middleware.
The high-level architecture of the Middleware communicating with the hardware or simulation and higher-level subsystems provides a solid framework that is easily extendable with further components. Future work may include *Collision Avoidance* for moving obstacles, e. g., other boats, and an *Emergency Behavior* to cope with a loss of signal.
The implementation has been tested with manually created sample data. To further examine the applicability of the approach, real world experiments with the boat need to be conducted.

6 Workload

The following table indicates the tasks each team member has focused on within the projects execution. It also contains a rough time estimation each team member has invested.

Team member	**Tasks**	**Time**
Martin Endreß	JAVA implementation of the navigation system as annotated in java classes and git commits, Introduction to Navigation (3.1), Task Representation (3.3.1), D* Algorithm (3.3.2), Discussion (**??**), Poster final design.	Estimation: meetings + |significant commits|$*$ commit effort $\approx 30h + (78 * 0.6) * 4h = 217.5h$
Tobias Heckel	JAVA implementation of the navigation system as annotated in java classes (often implemented together with Martin), Task Representation (3.3.1), Integration into Middleware (3.3.3), Evaluation (3.4), Discussion (**??**)	Estimation: 7h/week (project & team meetings, at home) $*15 + 60h$ (in lecture-free time, project report) $\approx 165h$
Ostap Kharysh	wind expert (2.4.3), rotation expert (2.4.4)	
Yaryna Korduba	simulation (2.5)	
Ralf Lederer	JAVA implementation of the middleware system (software architecture, message passing, command interpreting & processing, sensors, experts, testing), communication protocol, local position encoder, concept for position expert. Text sections: (2.x) except (2.4.3), (2.4.4) and (2.5)	Estimation: 15 weeks * (10 hours work at home + 2 hours meeting) + 40h (project report, researching) ≈ 220h
Alexander Löflath	Navigation theory and early implementation, Controller theory and implementation. Project Report Sections: Chapter 4	$\approx 160h$
Jan Martin	Navigation theory and early implementation, Controller theory and implementation. Project Report Sections: Chapter 4	$\approx 170h$

7 Appendix

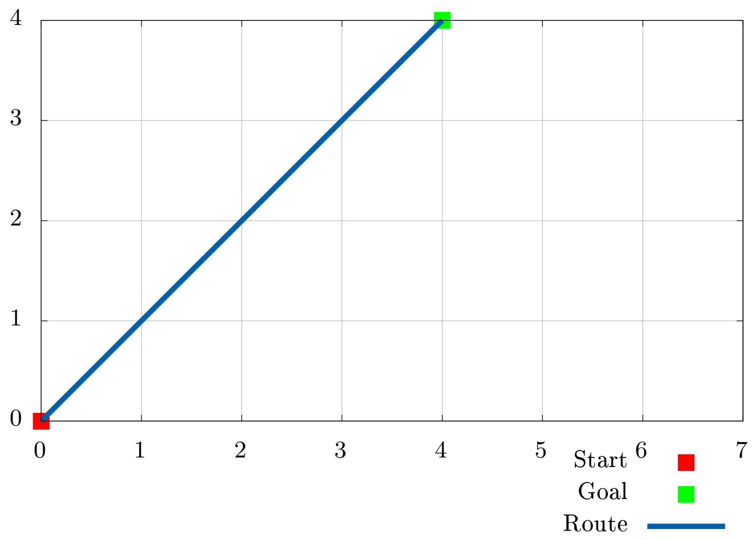
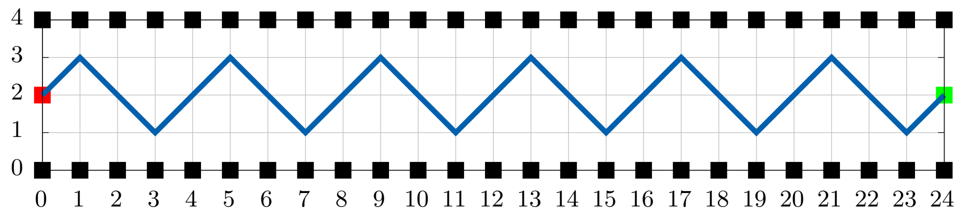
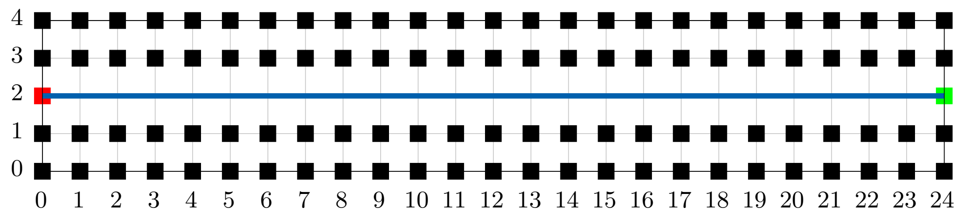


Fig. 28. *goodWindConditions*. Route in optimal wind conditions. The wind is coming from 333°(NNW). The heading of the boat towards the goal is 45°. The boat has the highest speed for this combination of wind and heading (see figure 15).



Obstacles	⬛
Start	🟥
Goal	🟩
Route	━

Fig. 29. *narrowMap*. Route in a narrow channel where the boat has to tack. The wind is coming from 90°(E).



Obstacles	⬛
Start	🟥
Goal	🟩
Route	━

Fig. 30. *narrowMap2*. Route of a narrow channel where one cell of the map stretches over the full width of the channel. The wind is coming from 90°(E). The boat needs to tack but it can't be represented.

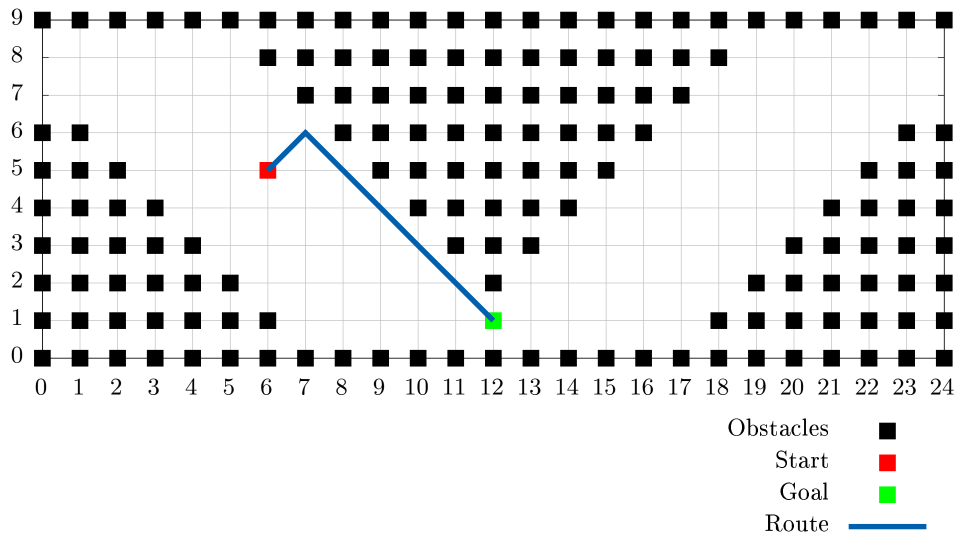


Fig. 31. *vTurn1*. Route in a narrow channel. The wind is coming from 90°(E).

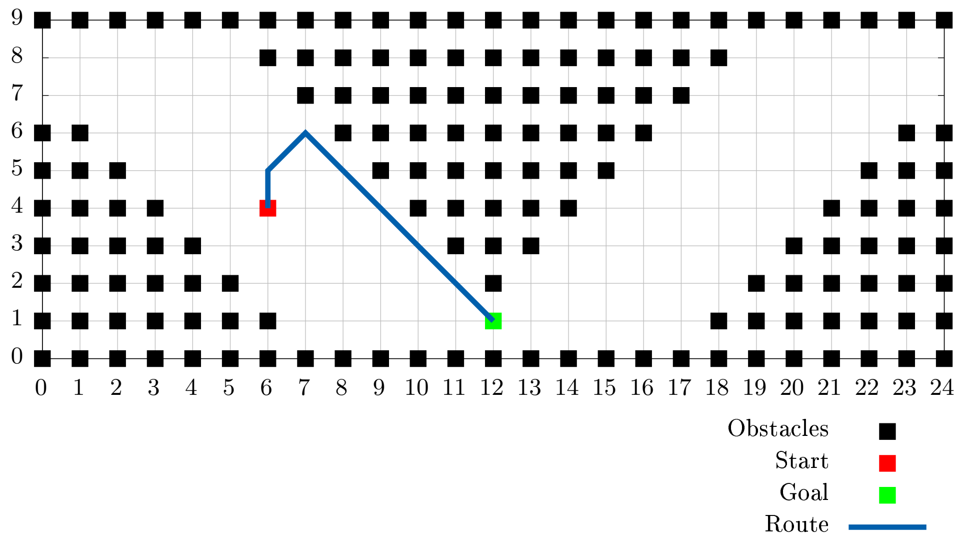


Fig. 32. *vTurn2*. Route in a narrow channel. The wind is coming from 90°(E).

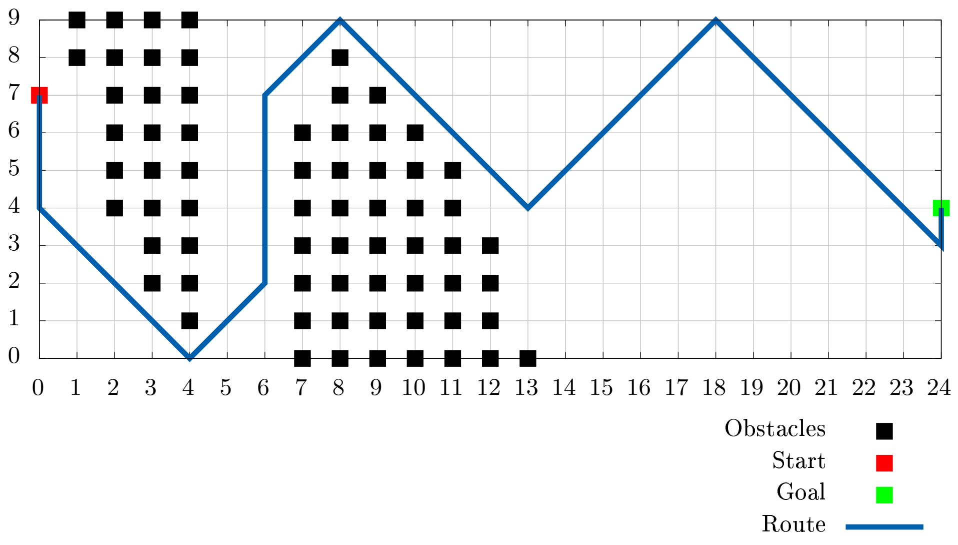


Fig. 33. *twoIslands*. Route through two islands where the boat has to tack. The wind is coming from 90°(E).

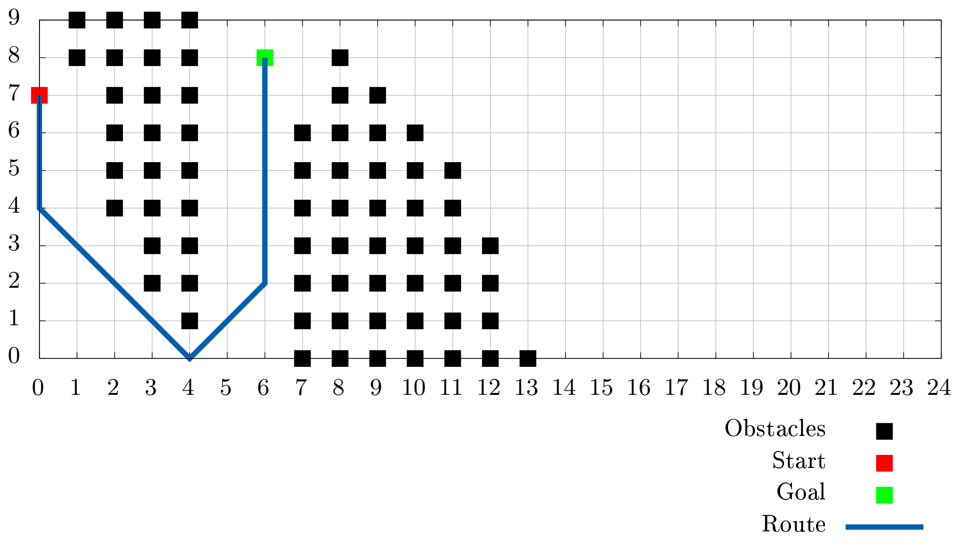


Fig. 34. *twoIslands-short*. Route through two islands where the boat has to tack. Compared to figure 26 the goal is nearer to the start. The wind is coming from 90°(E).