FH Dortmund

University of Applied Sciences and Arts

Department of computer science

Bachelor thesis

Estimation of fuel consumption of pleasure vessels based on environmental data models

DOI
https://doi.org/10.26205/opus-3198

Antonia Müller-Baumgart

supervised by

Prof. Dr. Sebastian Bab

Prof. Dr. Andreas Harrer

Dortmund, 18 May 2022

Abstract

A ship which cannot control its movement is unable to maneuver and might end up in difficult or dangerous situations. Therefore, it should be ruled out that such situations occur just because too little fuel was taken along.

The needed fuel for a planned route depends on the vessels consumption as well as on environmental influences, such as current or wind. Therefore many aspects have to be taken into account while calculating the needed fuel.

This document is about the question how the required fuel can be calculated in acceptable time and with devices which are suitable for vessels. First, the theoretical basis and a method was developed to calculate the fuel of a single section on a route. This method was used for different approaches of calculating the fuel over longer distances later. As input for environmental influences GRIB2 files where used.

The first approach was to calculate a path given by a user. Then an A* based algorithm was developed to find the path with the lowest fuel consumption between two points on the map. Later a concept was proposed to improve the implementation of the A* to a more universal approach in order to be able to include more aspects into the calculation.

It was shown that the implemented algorithms work well on constrained computation devices in a sufficient time range. A basic graphical user interface for testing purposes and a possibility to export the results into other navigational applications where developed. Further improvements, extensions and shortcomings of the available environmental models where discussed.

Zusammenfassung

Die Manövrierunfähigkeit eines Schiffes ist in jedem Falle zu vermeiden. Ein Schiff, das seinen Kurs und Geschwindigkeit nicht kontrollieren kann, läuft Gefahr in ungünstige oder gefährliche Situationen zu geraten. Daher sollte bei der Kursplanung ausreichend Treibstoff eingeplant werden, um die Wahrscheinlichkeit des Eintretens derartiger Situationen zu minimieren.

Der Treibstoff, der auf einer Route benötigt wird, hängt von dem Schiff, sowie von Umwelteinflüssen, wie Strömung oder Wind, ab. Daher müssen bei der Berechnung ggf. vielfältige Aspekte beachtet werden.

Diese Arbeit beschäftigt sich mit der Frage, wie der benötigte Treibstoff in akzeptabler Zeit auf geeigneter Hardware für maritime Umgebungen berechnet werden kann. Zunächst wurde die Theorie und danach ein Verfahren zur Berechnung des Treibstoffes pro Streckenabschnitt entwickelt. Dieses Verfahren wurde dann im folgenden für verschiedene Ansätze zur Berechnung des Treibstoffes über längere Distanzen verwendet. Als Quelle der Umweltdaten wurden GRIB2 Dateien genutzt.

Der erste Ansatz bestand darin, einen von einem Benutzer vorgegebenen Pfad zu berechnen. Danach wurde ein A* basierter Algorithmus entwickelt, um den Weg mit dem niedrigsten Kraftstoffverbrauch zwischen zwei Punkten auf der Karte zu finden. Später wurde ein Konzept vorgeschlagen, um die Implementierung des A* zu einem universelleren Ansatz zu verbessern, um mehr Aspekte in die Berechnung einbeziehen zu können.

Es konnte gezeigt werden, dass die Berechnungen der Algorithmen in akzeptabler Zeit auf der begrenzten Hardware zu zufriedenstellenden Ergebnissen führt. Es wurde eine einfache graphische Benutzeroberfläche entwickelt welche u.a. die Möglichkeit bietet, die Ergebnisse in andere Anwendungen zu exportieren.

Contents

List of Figures

Glossary

AIS	Automatic tracking system for collision prevetion (long: Automatic Identification System)	25
at anchor	A ship which is attached to the ground with an anchor or similar.	12, 29
broaching	The ship turns sideways to the waves and is pushed onto its side.	13
compass	Either a drawig tool or a tool which shows the magnetic direction	36
disabled	A ship which has limited abilities to manouver.	12
drift	Effect if a ship is moved to the side by an environmental influence	5, 12, 34–36
dyke	A long wall or similar to protect a landmass from beeing flooded.	72
head to wind	A course directed straight against the wind.	21
latitude	Imaginary lines parallel to the equator	17, 19, 20, 45, 51

longitude	Imaginary lines from the north pole to the south pole	17, 19, 20, 45, 51
lull	The condition of very less or no wind.	13
moored	If a ship is attached to something like an anchor, bouy or jetty it is moored.	12, 29
mud flat	Coastal areas which are areas of land on low tide or covered by water on high tide e.g. the "Wadden Sea"	71
polar diagram	A diagram which shows the speed of a vessel under sails depending on wind speed and angle to the wind.	79
rudder flow	The rudder of a ship can take effect if there is a water flow against the rudder blade. This condition is called "rudder flow".	12
run agound	A ship which drove into shallow water so it got stuck on the ground.	12
sail	Foreward movement of a ship. A ship sails, even if it has no sails and is driven by a engine.	5, 12, 13, 21, 22, 24, 36, 63
sparse file	A file in which 'empty' parts are not stored. As an example a file with content in the beginning and the end and a empty part in the middle is not filled with padding zeros.	17

tide	The rise and fall of the sea because of the gravitation of the moon. The time between high tide (highest water level) and low tide (lowest water level) are around 6 hours. There are ~4 tides a day.	13, 63, 68, 71, 74
to bunker	To fuel a ship.	13
to motor	To use the engine to drive the ship mainly or in addition to the sails.	13
towed	A ship is towed if it is pulled through the water by e.g. another ship.	13
under sail	A ship which drives with its sails only. A ship which has the sails up, but has the engine running is ***not*** under sail, but under motor. The engine does not have to drive the ship actively (e.g. engine in standby, clutch not engaged).	12, 14, 22, 79
underway	A ship which is neither moored, at anchor nor run on ground.	12, 29
upwind	All courses where the wind comes not exactly from the front, but a little bit from the side until an angle nearly from the side. The wind exactly from the side or even more from the back is no upwind.	21

Acronyms

bft	Beaufort, commonly know as the wind force.	20
COG	Course over ground	21, 36
CTW	Course through water	21, 36
DOG	Distance over ground	21, 35–38, 50
DTW	Distance through water	5, 21, 36–38, 50
ft	feet, $1ft \approx 0,3m$	20
hp	Horse power, $1hp \approx 0.75kW$	34
kn	knot, NM per h ($kn = \frac{NM}{h}$), $1kn = 1.852\frac{km}{h}$	13, 20, 21, 36, 39, 40, 68
NM	nautical mile, $1NM = 1.852km$	20, 21, 36, 51, 67, 68

rpm	revolutions per minute	33, 65, 81
SOG	Speed over ground	21, 50
STW	Speed through water	21, 36–38
WMO	World Meteorological Organization	15

Chapter 1

Introduction

In the maritime sector a ship is said to be “underway” if it is not moored to shore, is not at anchor and has not run agound [kvr09]. In other words, a ship is underway, so moving when it is not connected to a landmass, even if it is not actually moving. This is because a moving ship has to follow traffic rules and must therefore make sure that it can for example avoid collisions with others. If a vessel is unable to ensure the former because of defects or the like, it has to mark itself as “disabled” to warn others.

If a motorboat runs out of fuel like diesel, petrol or electric power, it can no longer maneuver. A sailing ship can still sail, but only under suitable conditions. It needs wind of the right direction and strength. Without wind it cannot sail and in a storm the sails might need to be taken down. A sailing boat cannot go against the wind and harbors often cannot be entered under sail because of the constrained space to maneuver or it might be simply forbidden.

A ship without propulsion can drift away, either to the open sea or towards the coast and thus run agound or crash on a cliff. On busy routes, the ship can collide with other ships, for example a tanker, and sink as a result. Maritime traffic areas are often very crowded on narrow routes or near ports and not necessarily designed for a sailing boat to be able to take evasive action in these situations.

In a storm, a ship without a drive and thus rudder flow cannot sail against the waves to avoid broaching (the ship turns sideways to the waves and is pushed onto its side). Even if the ship is stationary in a lull and no current, other supplies such as water and food might run out after a while.

These cases emphasizes that a vessel should never run out of fuel to avoid complications or worse. If a ship has to be towed because it is out of fuel it might get expensive as most insurances might not pay such cases. In addition to the cost factor, being on the water without fuel is a situation most boaters would like to avoid most likely as it might become very stressful for the crew.

Calculating the required fuel exactly for a route might become complex if it depends on many factors such as speed, wind, waves or current. This is the motivation to develop an application, which calculates a ship's probable fuel consumption while taking environmental factors into account.

1.1 An example use case

For better understanding and as an example use case the last trip of my crew is used. The goal of the trip was to sail from the Netherlands to England, or more precisely to a British pub named "The Butt & Oyster". The pub is famous among sailors from the area, as it is a common challenge to sail to England and back in one long weekend to this special pub.

The boat used was a small sailboat of 30 feet (approx 9m), which can drive 5kn in average. So the question was "how much fuel will this special boat need on this tour?".

In the channel between the UK and Europe are mostly currents which will hit the vessel from aside, depending of the tide. Therefore the ship has to be steered against the current most of the time. It was not much wind expected at this particular weekend so it might be necessary to motor the whole way.

Fuel was very expensive in the UK at this time due to the Brexit and the Russian invasion of Ukraine in 2022. Therefore the plan was to bunker enough fuel to avoid the necessity to bunker in the UK again. As fuel was expensive in the

EU anyway and diesel tends to rotten on ships (diesel bug, diesel pest) leftovers should be avoided. To estimate the fuel consumption became tricky as it was unclear how much of the trip will be covered under sail.

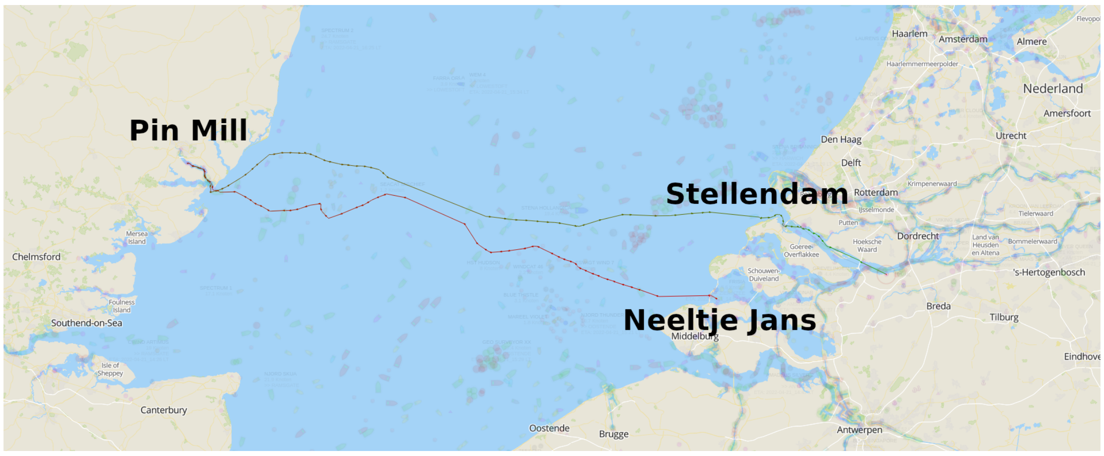


Figure 1.1: The graphic shows the track of the sailboat on the way from the Netherlands to England and back. The trip started at the lock at Neeltje Jans and headed to Pin Mill (red line), a small village near Ipswich. The way back (green line) headed to Stellendam because of the wind at that day.

This leads to the main question of this thesis: How can the fuel consumption of a specific ship be calculated taking environmental influences into account? Furthermore can it be calculated in a usable time range and with devices which can be run and powered on board of a pleasure vessel? These are the questions which will be discussed in the following chapters.

Chapter 2

Terms and technologies

This chapter describes terms and technologies used or referred to in this document. This includes both, technical and nautical explanations.

2.1 GRIB files

GRIB stands for *General Regularly-distributed Information in Binary form* and is a standard by the World Meteorological Organization (WMO) for meteorological data. The format allows to save gridded data points matching geographical points and time. Commonly known are weather forecast maps which display the wind direction and speed for example, but the format can be used for much more use cases.

There are three versions of GRIB, in the following GRIB will always mean GRIB2, as this is the newest version at the time. The following description of GRIB messages are based on the *Guide to the WMO Table Driven Code Form Used for the Representation and Exchange of Regularly Spaced Data In Binary Form: FM 92 GRIB Edition 2* [gri03]. All code tables can be found in the *Manual on Codes* [WMO21].

Each GRIB message is divided in 9 sections, of which one is optional. The sections of a GRIB2 message are numbered and named as follows:

- Section 0: Indicator Section
- Section 1: Identification Section
- Section 2: Local Use Section (optional)
- Section 3: Grid Definition Section
- Section 4: Product Definition Section
- Section 5: Data Representation Section
- Section 6: Bit-Map Section
- Section 7: Data Section
- Section 8: End Section

All sections must appear at least once in every GRIB message except section 2 *Local Use Section*. Some sections can be repeated in a single message. This allows more flexible messages, for example a single message can contain a forecast for different periods of times by repeating section 4 to 7 for each time window.

GRIB messages are build in a octet structure, each section ends at an octet boundary. To accomplish this, the sections get expanded with zeros if necessary. The length of the sections is variable so at the beginning of each section the first four octets contain the length of the current section in octets, followed by the number of the section. This applies for all sections except the indicator and end section, as these sections have a fixed length.

Basically every section defines several sets of octets referring to tables and the permitted values of them. The tables are long lists of definitions, each with a referring value.

The first three sections are used to save information about the origin of the message, the date it was created and the status of the message, if it is a forecast, a recording or an analysis for example. The sections 3 to 6 serve to describe what kind of data are present and how they can be interpreted, while the data section (section 7) contains the actual data.

The *grid definition section* defines as the name indicates the grid surface and geometry for the following data. This grid can be either defined by reference to one of the grids defined in the GRIB standard or be a reference to a proprietary definition not part of the GRIB, but most messages use a standard latitude/longitude grid.

The *product definition section* describes the meaning of the data in the message. Among other things the section contains the category and number of the message, together with a parameter 'discipline' from the *indicator section* they define the type of data presented.

The *data representation section* defines the format the data is saved. To indicate the presence or absence of data values at each data point, the *bit-map section* uses a bit string with one bit for each data point, ordered like defined in the *grid definition section.* If a bit is zero, the corresponding data point has no value and is not part of the GRIB data (like a sparse file).

With the information of these sections the actual data in a GRIB message can be interpreted. The grid defines **where** on the globe each data point refers to, the product definition tells **what** the data represents (wind, current, etc.). The data representation describes **how** the data is stored and the bit-map defines **which** data is missing.

2.2 Open Plotter

Open Plotter is a navigational aid for small and medium boats. It consists of several software and hardware components which can be combined freely to meet with the individual requirements of each vessel. The project is Open Source and works on ARM computers like a Raspberry Pi. [Ope21]

Open Plotter includes a chart plotter, offers the possibility to build instrument panels to visualize data and data can be send and received on different ways, including the maritime standards NMEA 0183, NMEA 2000 and different wireless solutions. It can connect to multiple sensors and display GRIB files, mostly used for weather forecasts.

2.3 Search algorithms

This section describes basic search algorithms and concepts, which are used or mentioned in this thesis. The search strategies mentioned in this section refer to tree and graph based searches.

2.3.1 Heuristic

Search strategies which depend on specific knowledge of the problem can find solutions more effectively. Most algorithms use a heuristic function for this purpose. A heuristic function or short heuristic estimates the costs for each node to the target. The better the heuristic is, the more efficient the search algorithm will become. [SN16][1]

Heuristics can have various conditions. An often required condition is admissibility. An admissible heuristic never overestimates the costs to reach a goal and is therefore optimistic. This is important, as search algorithms might miss better solutions if otherwise. [SN16][2]

2.3.2 A* search

A* is a search algorithm which uses the cost to reach a node and the cost estimated by the heuristic to reach the goal from the node. The algorithm is complete and optimal, so it will find a solution if it exists and it will be the best solution, if following conditions are fulfilled:

The heuristic has to be admissible to make an A* search optimal, otherwise the algorithm might leave out important ways in some cases.

If the search is applied to a graph search the heuristic also has to be consistent. Consistence means that for every node n (and all it successors) the estimated costs are not greater than the costs to get to the next node plus the estimated costs to the goal of this node. [SN16][3]

[1]page 93
[2]page 94
[3]page 95

2.3.3 Bidirectional search

A bidirectional search starts a search from the start and endpoint simultaneously aiming to meet in the middle. With increasing depth of a search tree, the time and space complexity usually also increases exponentially. If two search trees are used instead of one, both search trees are only developed to half the depth and are therefore consuming less time and space together than the single search tree. The solution may not be optimal, but can be found in a short time. [SN16][4]

A bidirectional search is not always applicable, as to search from the end, the end state(s) must be known. If the end is described as an abstract description, it might be difficult to use such a strategy.

2.4 Nautical terms

As this thesis is about a maritime system, some nautical terms will occur. This section provides a basic overview of these nautical terms.

2.4.1 Coordinates on the globe and compass courses

As there are no or very less buoys, landmarks and alike on high seas navigation is mainly based on compass courses and GPS coordinates nowadays. Compass courses are often named after the 8 main directions (north, north-east, east, etc.) or indicated in degrees (most commonly in $0° - 359°N$).

Latitudes are imaginary lines parallel to the equator and counted from $0° - 90°N$ or $0° - 90°S$, with $0°$ is exactly on the equator. Longitudes are imaginary lines from the north pole to the south pole and is counted from $0° - 180°E$ and $0° - 180°W$ with $0°$ is located in Greenwich, London. Each degree is subdivided into 60 minutes and each minute into 60 seconds. The distances of this units are not uniform on the map as the distortion and the distances between the longitudes depend on the position on the globe. [Sle19][5]

[4]page 90 - 91
[5]page 318 - 319

Geographical coordinates are commonly given as a pair of latitude and longitude, so the coordinate of the *FH Dortmunds department of computer science* is 51°29'37.3 N 7°25'13.0 E for example.

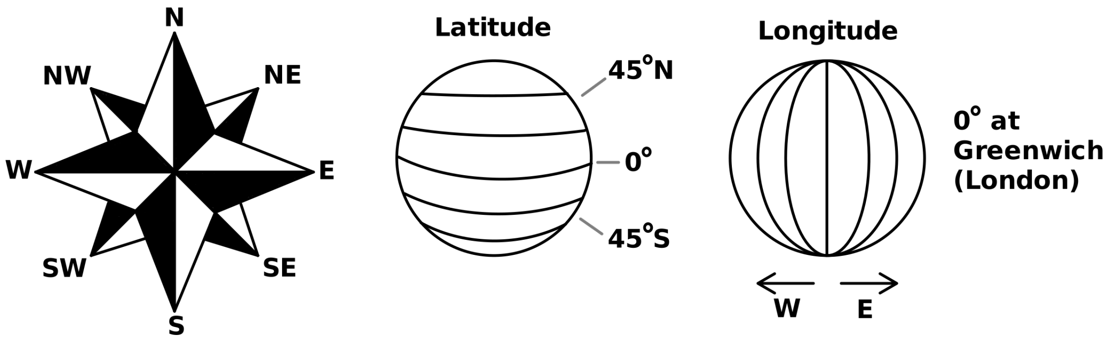


Figure 2.1: Compass courses, longitudes and latitudes on the globe

2.4.2 Nautical units

In the maritime sector many non-metric units are used, like the length of vessels is often referred in feet (ft) and wind is measured in Beaufort (bft), a scale from 1805 which describes the effect of the wind to the sea and to the sails of a frigate [Off10]. The wind force commonly given in the news (e.g. "storm with wind force 9") refers to the Beaufort scale.

In the following the units NM and kn will be used for distances and speed mainly. NM stands for nautical mile and was defined as one minute of arc on the equator, so $\frac{1}{60}$ of one degree (latitude). Today it is defined based on the SI-System as $1,852km$. [BidpemoWM77][6]

Knots (kn) is defined as NM per hour, so $1.852km/h$.

2.4.3 Speed over ground and speed through water

A major difference between vessels and vehicles like cars or trains is the ground on which they move. If a car drives it will make the corresponding distance over

[6]page 12

ground, but if a boat drives through the water, it does not necessarily travel a distance over ground.

This is why a distinction is made between *speed over ground* (SOG) and *speed through water* (STW) for vessels. STW indicates the speed of the boat compared to the surrounding water while SOG is the speed seen from land or any other fixed point. [Bar90][7]

Similar to the concept of STW and SOG, other units are distinguished too. The *course over ground* (COG) is the course which the vessel drives according to a fixed reference like a GPS position, while the *course through water* (CTW) is the course which the vessel is heading to, which may differ in some situations. Same goes for distances, the *distance over ground* (DOG) is the distance covered according to a fixed point similar to the SOG. The *distance trough water* (DTW) on the other hand, is the distance covered in the water.

> *If a vessel maintains a CTW of* $0°N$ *and a STW of 5kn for one hour against a south current of* $180°N$ *with a speed of 5kn, it will have driven a COG of* $0°N$ *and a SOG of 0kn with a DTW of 5 NM, but a DOG of 0 NM after one hour. It might feel like long trip at high speeds for the crew but the ship never left the spot.*

In the maritime sector most of the navigational units are distinguished into *through water* and *over ground* and sometimes also a "magnetic" component occurs, like the *compass course* vs. the *true course*, but such terms do not appear in this document and are mentioned for completeness and to spark curiosity by the reader.

2.4.4 Sailing in different angles to the wind

Depending on the angle to the wind (course to wind), sailboats have to configure their sails to be able to sail different courses (CTW, COG, etc.). Modern sailboats may be able to sail upwind courses (a course on which the wind comes nearly from the front), but no sailboat can sail head to wind (directly into the wind)

[7]page 44

under sails.

Depending on the angle to the wind, sailboats can sail at different speeds. It depends on the design of the ship, but in general it can be said that sailing ships with wind from the side are faster than with wind from diagonally in front or exactly behind. [Sle19][8]

[8]page 89

Chapter 3

Related work

This chapter shows projects and papers which deal with related topics or use similar techniques. The paper *Ship weather routing: A taxonomy and survey* deals with different approaches for ship weather routing in order to optimize routes. The *Long-term routing for autonomous sailboats* is an approach to route a autonomous sailboat under the use of weather GRIB files. In *Projects implementing weather routing* two projects are displayed, which use weather data to suggest optimized routes for sailors.

3.1 Ship weather routing: A taxonomy and survey

The paper *Ship weather routing: A taxonomy and survey* [ZPD20] gives an overview of the developments in the field of weather routing. The paper states that most of the studies are related to container ships and bulk carriers.

The main difference between ship routing and ship weather routing is the problem category. Ship routing deals with questions about which vessel should be when at which harbor to load and unload which goods, a similar problem to the traveling salesman problem [HK70].

Ship weather routing is about optimizing aspects of a specific ship which shall sail from a specific point A to point B, based on environmental influences such as wind, waves or currents. The optimization can aim at different goals, like fuel consumption, safety or punctuality.

In theory, ships should use the least fuel when traveling at a steady pace, as accelerate and brake are most fuel consuming. But in reality this is often not happening because of crowded areas or timings.

The paper distinguishes four main categories of approaches for weather routing: The modified isochrone method, dynamic programming, pathfinding and genetic algorithms and artificial intelligence and machine learning.

With the isochrone method each isochrone is a line, which represent a possible trajectory of a vessel. At the end of each line a new one begins until the goal is reached. With the modified method also environmental factors are included, so the length of the lines can differ. Dynamic programming follows the idea of dividing a problem into many stages, so that after deciding the first decision, all others must be optimal.

Many studies relay on pathfinding algorithms like Dijkstra or A* and include different factors of weather routing. There is a wide range of approaches and customized versions of these algorithms to find an optimal route, mostly focused on environmental and cost factors. Newer artificial intelligence approaches focus mainly on predicting fuel consumption under different conditions, either to optimize on-time arrivals or fuel saving.

The paper states that average fuel savings with weather routing approaches are between 3 to 5%. Some paper report fuel savings up to 19%, but due to a missing standardization on both the different vessel types as well as different areas on the globe the results of different papers are difficult to compare.

3.2 Long-term routing for autonomous sailboats

The paper *A Rule-Based Approach to Long-Term Routing for Autonomous Sailboats* [LSF11] describes an approach of a routing algorithm for sailboats, which adapts the route to the latest weather forecasts constantly. The goal is to find the fastest way between two way points. To do so, it uses wind speed and direction from GRIB files in combination of the speed of a sailboat depending on the angle to the wind.

To be able to apply classic search algorithm to the infinite number of points in the sea, the search space was gridded into small fields. Fields located on land are excluded from the graph based on a binary map. Each field is annotated with a wind vector from a forecast.

To find the best route an A* approach is used. The heuristic function uses the distance between the current position and the goal and divide it through the hull speed of the vessel. This creates a consistent and cheap to calculate heuristic.

3.3 Projects implementing weather routing

There are several weather routing applications available. Most of the applications are designed for the commercial sector and big vessels like container ships and not for small pleasure vessels.

OpenCPN is an Open Source chart plotter software for small vessels in contrast. With its plugin *Weather Routing Pi* [DEp21] it supports weather routing too.

The weather routing plugin uses the isochrone method, GRIB files and climatology data to calculate the ideal route. Each isochrone represents how far the vessel can drive in a given time slot. To calculate the weather influences the GRIB data is used if available and climatology data as fallback.

SailGrib WR [sai21] is a popular commercial routing app for sailors. It can calculate the fastest route based on weather, current and boat characteristics. It includes many useful features like AIS or sea charts, but it is closed source and provides no fuel calculating feature.

Both applications may use GRIB files for routing, but are more focused on optimizing routes under sails and avoiding dangerous areas like hurricanes than to optimize the fuel consumption. The Weather Routing Pi is furthermore more designed for experienced sailors than for beginners, the author described his tool as “[...]remarkably flexible, but with that comes complexity” [DEp21]. Therefore a tool like this might overwhelm beginners.

Chapter 4

Project

This chapter describes the project with all its aspects. It begins with the section *4.1 Requirement analysis* followed by *4.2 Concepts*. The requirements are very closely related to the conditions on ships and the shipping environment itself, the concepts provide solutions to comply with the requirements.

The section *4.3 Theory* explains the theoretical foundation the *4.4 Implementation* is based on. The theory deals with topics such as how the fuel consumption of the ship is determined or how the current and wind affect the ship. The implementation shows how the final algorithm works and how it was implemented.

4.1 Requirement analysis

This section explains the requirements the project is based on. The requirements are based on conditions on a vessel and possible use cases.

The application needs to be usable offline and work in remote areas which leads to the requirement *4.1.1 Universal data sources*. The next requirement *4.1.2 Ease of use* explains in which aspects the system have to be reliable in certain situations.

In *4.1.3 Time vs Quality* the weighting of the time in which a result is available versus the quality of the result is discussed. The requirements for the way the results are presented are explained in *4.1.4 Output as map*. Lastly the reasons for using a single board computer and open source software are explained in *4.2.2 Single board computer* and *4.1.6 Open Source.*

4.1.1 Universal data sources

The area in which ships operate tend to be remote and reliable data sources like internet are rare and expensive. The system should rely on devices and sources on the vessel only. It should work offline without any external connections.

Data inputs should be designed universal a based on common standards to ensure compatibility to as many data sources as possible.

It should be easy add or update data if accessible. It should be easy enough to be applied in every target destinations with a data source (e.g. internet) like harbors or similar. Public data sources like satellite broadcasts or regular VHF radio data transmissions should be usable if compatible to common standards and applicable. The more universal the data input is designed, the more flexible the system will be.

4.1.2 Ease of use

The system is a tool which assists in course planning. Thus it should less stand in the way than it helps. This means it should require less to no configuration and as less maintenance as possible.

Maintenance should focus on data update mainly which should be easy, as it is a regular task most likely. It should work out of the box without major initial installation and configuration to make it suitable for unexperienced users.

4.1.3 Time vs Quality

Even if it would be desirable for the system to provide hight quality results in a minimal time frame, the complexity and thus the computational constrains might make it impossible to provide a solution to satisfy all desired results at the same time.

Therefore a weighting of preferences depending of the situation will be desirable to meet the most preferred goals.

The analysis of the following scenarios provide two complementary use cases:

Scenario one is if the vessel is not underway (moored, at anchor, in a harbor, etc.). While course planning for the next day time efficiency is not that important, but the user is interested into a detailed result. A calculation might run over night and still meet the requirements of the user.

Scenario two in contrast takes place on sea when bad weather occurs. In this situation the user might be more interested in coarse but fast results. Precision is a less important factor but it might be desirable to check a few different routes to harbors around quickly.

These scenarios describe two situations opposite in their requirements. As can be seen in this example the preferences into time and quality should be customizable to conclude with the requirements of the situation.

4.1.4 Output as map

The result of the calculated fuel consumption should be displayed as a map. This makes it easy to read the results and adapt the information to the existing course planning. The map can display the calculated fuel with color, as this is intuitive for most people.

With the use of a colored map the user can compare several results more easily. A user can see which areas are particularly interesting, for example, if a current between two islands is particularly strong and the ship is accelerated or braked as a result.

In addition to the colored map, the outputs should of course also be available as numerical values, such as the fuel required over the entire route or the travel time. Beside the absolute values of the calculation the colored map provide additional information, which leads to *Output as map* as an requirement.

4.1.5 Limited resources

As resources like space and power are very limited on vessels, especially on long term any hardware like the devices for the computation need to comply with this constrains.

On pleasure vessels 12V or 24V supplies are very common. If 230V are needed additional hardware is required which is still very constrained if provided in most cases. The hardware should not exceed a consumption of more than 15W, as this is the usual range of consumption for long running systems on vessels.

As space is a very limiting factor the hardware should be as small as possible and should not extend the size of a 'shoe box' at maximum.

4.1.6 Open Source

As Open Plotter is the most popular project in this field, the project should fit with Open Plotter. A compatibility with Open Plotter would have several advantages, like a communication interface to other ship systems or sensor data.

As Open Plotter and the software surrounding it is Open Source, the source of the project should become Open Source too.

If even a community forms around a project this community may improve the project by testing, finding and often solving bugs and in the best case ensures even a constant further development of the project.

4.2 Concepts

To meet the requirements several concepts are developed, which build the foundation of the implementation. In *4.2.1 Interfaces* the use of a universal data input is discussed. The concept of *4.2.2 Single board computer* fulfills the requirement of constraint power and space.

To meet different preferences of runtime and quality, different algorithmic concepts are discussed in *4.2.4 Various solution patterns* and *4.2.3 Staggered calcula-*

tions. With the concept of *4.2.5 Separation of concerns* modularity and therefore good code quality should be ensured to make the project reusable and improvable like stated in *4.1.6 Open Source*.

4.2.1 Interfaces

A highly important concept is the use of interfaces. Apart from the fact that the interfaces can bring a certain order with them by requiring structured code and hardware, interfaces make a system modular as single components can be exchanged easily because of clearly recognizable dependencies and separation of concerns.

A good interface is the use of GRIB files as input data for weather models such as wind and stream charts. GRIB files a highly common in nautical navigation and therefore available from various sources. This gives the great opportunity to use data complete independent of their origin, which complies with the requirement of *4.1.1 Universal data sources*.

4.2.2 Single board computer

Due to the lack of an internet connection onboard an outsourced calculation on a online server returning the results to a smartphone or other device is not possible. A onboard server or a laptop are both not suitable, as space and power are limited on a vessel and therefore bigger installations or power hungry devices may be not suitable especially on smaller vessels.

A common Open Source solution is the usage of a Raspberry Pi or a similar small board computer on pleasure vessels. It is used as the main system for sensors, navigational tasks, automatization or dashboards often. The software developed in this project should be suitable to run on such systems.

The very common Raspberry Pi 4 8GB draws less than 15W on average [Ltd19][1], is small enough to fit in very limited space and meets the requirement of

[1]chapter 4.1

4.1.5 Limited resources this way. It is assumed to provide enough computational power for the project.

4.2.3 Staggered calculations

Depending on the situation different amounts, types or quality of input data may be available for the system. To achieve the flexibility required in *4.1.1 Universal data sources* the system should be able to process the different inputs. It would be desirable to combine the different inputs in a uniform model.

This could be achieved by stacking the results on each other and base calculation on previous results if possible. It could be used to make the map more detailed with each additional input or if detailed data is available later. The data can be added to an already existing model without start calculating from scratch.

> *If a fuel consumption map based on a GRIB containing currents was already calculated and the user wants to take also a GRIB containing wind data into account, the calculation could run more efficient if the second 'wind' calculation is based on the already calculated 'current' based data.*

4.2.4 Various solution patterns

As stated in *4.1.3 Time vs Quality*, different situations may lead to different requirements regarding the quality of the results and the time needed for the calculation to provide results. To comply with these requirements various solution approaches are used.

The most intuitive pattern is to start at a point and develop the map into all directions evenly. With this method every reachable point will be calculated and maybe even ways found the user did not think of, but as it calculates every point on the map, it might be to time consuming to suit the situation.

To save some time, only fields of interest could be calculated. These fields of interest would have to be selected by the user. This might be a start and end point, which would allow the use of heuristics or a bidirectional search. If more way points or even a complete path is provided, the used search algorithm can find a solution even faster than an algorithm which searches in all directions to cover all solutions.

4.2.5 Separation of concerns

In order to achieve modularity of the code base and a good structure of the functionalities the development should follow the concept *divide and conquer*. This means that every task should be isolated and implemented a separate function if possible. By doing so, is becomes much easier to exchange parts of the code in order to improve or to implement new functionalities, extend the code and refactor the project if necessary.

Beside the exchangeability, it increases the readability if suitable names are used for each function. The development becomes more focused to the main task as a good structure tends to lead to a lean solution in most cases.

Following this concept increases the code quality and makes it possible to read the code even after a long time.

4.3 Theory

In this section the theoretical bases of the algorithms are discussed. To estimate the fuel consumption of a vessel over long distances the main factors are the average fuel consumption, the speed of the vessel and the impact of environmental influences.

4.3.1 Fuel Consumption of a vessel

The average fuel consumption of a vessel depends on several aspects. The size and form of the boat, the size and type of engine the state of the engine (rpm, temperature, etc.), the condition of the engine (e.g. state of maintenance) and the underwater ship (algae growth on the hull) and much more can influence the efficiency and consumption of a vessel. A rule of thumb says $0.21l$ diesel or $0.29l$

petrol per hour and per hp.

A more detailed estimation is provided by most manufacturers of engines. As an engine for a pleasure vessel is build for a special type and size of boat mostly, this estimation will also match to the ships dimensions roughly.

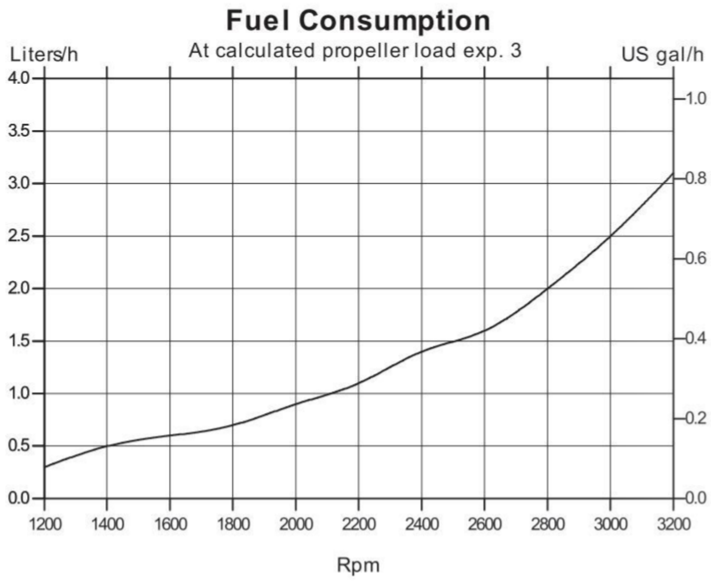


Figure 4.1: Exemplary the fuel consumption of a Volvo Penta D1-13. The consumption depends of the number of revolutions. [Pen16]

Under the use of such a diagram and a little bit of empirical knowledge a owner of a vessel can estimate the fuel consumption per hour of his boat.

4.3.2 Drift

A boat is a object moving in the two systems air and water and is therefore influenced by both. If the environmental influences like wind or current work against the vessels movement the vessel is slowed down. If they work with the movement of the the vessel it is speed up.

If the environmental influences work sideways to the vessels moving direction, the ship will drift to the side. The resulting course will be displaced by the angle (α in figure *4.2 Drift of a vessel*) which results from the triangle formed from the intended course and environmental influence. To prevent this, the helmsman has to counter steer against the environmental influence. [Sle19][2][Bar90][3]

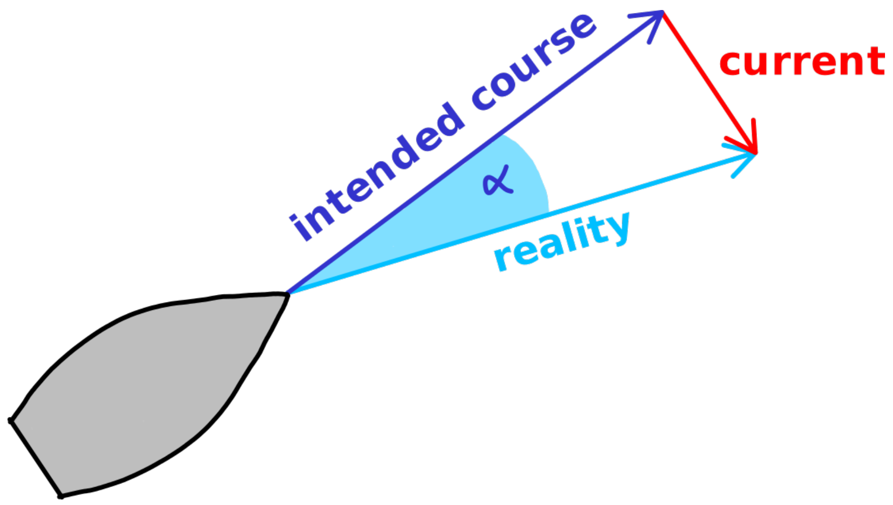


Figure 4.2: The vessel is driven away by a current by the angle α, the real course results from the intended course and the current. This phenomena is called *drift*.

4.3.3 Vectors and trigonometry

The heading of a vessel and most environmental influences such as wind, waves or current can be represented as vectors. The direction of a force is represented by the angle of the vector and the length of the vector represents the force (speed).

With this representation, different forces can be offset against each other easily. To prevent the vessel from drifting away, the angle to counter steer is needed. This angle depends on the environmental influence and the speed of the vessel.

In classic navigation using paper maps and compasses (drawing tool), the angle is calculated as follows: The DOG (distance over ground) gets marked on the map. Then the environmental influence, in most cases the current, gets also drawn into the map from the start point. Next the compass is set to the length equivalent to the speed of the vessel.

[2]page 348 - 349
[3]page 90 - 93

If the vessel sails with 5 kn, it goes 5 NM per hour so the compass would be set to the equivalent length of 5 NM of this map section.

The compass is pointed to the tip of the current vector and the radius is drawn to intersect with the line of DOG. With this the intersection of the speed and the DOG gets marked. With this point a triangle is created. The result can be seen at *4.3 Course correction and DTW of a vessel steering against the drift.*

With this triangle two things can now be determined: First the angle β is the angle by which the ship must correct its course to counteract against the drift. To gain the time which the ship will need to cover the DOG influenced by the current the DTW is needed. If a line which is parallel to the STW (line) is drawn, which meets with the target point and the (extended) line of the current, the length between the intersection of current and this line and the intersection of DOG and this line represents the DTW.

By simply measuring the length of this line and dividing it through the STW, the time needed for this course is found. [Bar90][4]

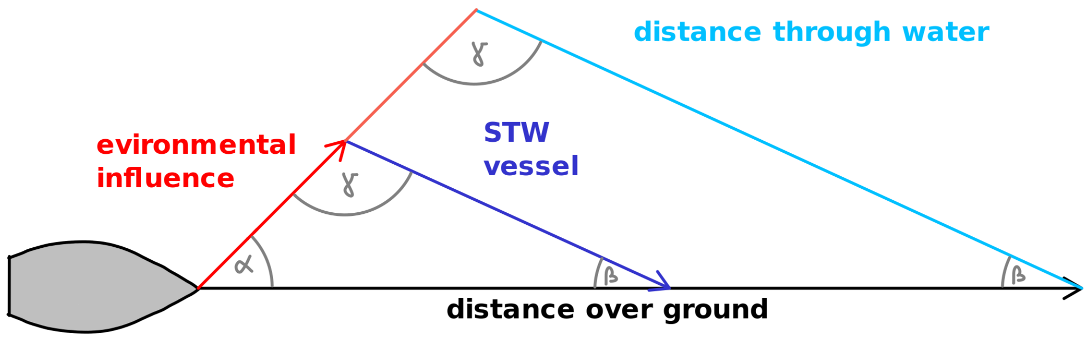


Figure 4.3: The smaller triangle is formed from the environmental influence, the COG (desired) and the speed of the vessel. The angles of this triangle are the same as those of the bigger triangle. As the DOG is known (as it is a desired parameter), the DTW, CTW and time can be calculated.

For the algorithm this problem will be solved mathematically instead of graphically. The angle α is the angle between the environmental influence ($\overrightarrow{env}$) and the

[4]page 94 - 95

vector which represents the course of the vessel ($\overrightarrow{course}$). The angle is calculated with:

$$\alpha = arcsin(\frac{\overrightarrow{env} \circ \overrightarrow{course}}{|\overrightarrow{env}| \cdot |\overrightarrow{course}|})$$

The angle β is calculated by:

$$\beta = arcsin(|\overrightarrow{env}| \cdot \frac{sin(\alpha)}{STW})$$

with the length of the environment vector ($|\overrightarrow{env}|$) and STW as the speed of the vessel. With the two angles, the third angle γ is $180° - \alpha - \beta$ as it is a triangle.

$$\gamma = 180° - \alpha - \beta$$

The extended triangle has the same angles as the first one and the length of one of the edges is known, as this is the DOG. With these information the DTW can by calculated with:

$$DTW = \frac{DOG}{sin(\gamma)} \cdot sin(\alpha)$$

The time of DTW is calculated with:

$$time \;=\; \frac{DOG}{STW}$$

The method works well except for two cases: If the environmental influence is nonexistent (no wind, no current, etc.) or has the exact same direction as the vessels course. In both cases no triangle can be formed to calculate the values, so these cases have to be processed differently. If the environmental influence head in the exact same direction, it will speed up the vessel by the speed of the influence. If the influence heads in the exact opposite direction, it will slow down the vessel by the influences speed. If the influence is non-existing, it will not change the speed or direction of the vessel at all.

All these cases can be solved by adding the vector of the vessel and the vector of the environmental influence, as the resulting vectors length is the STW of the vessel. With this information the time and needed fuel for the DOG can be calculated as in this case DOG and DTW are the same.

4.3.4 Calculation of the range

With the knowledge from the chapter *Vectors and trigonometry* the fuel consumption between two points with environmental influences can be calculated. Since even a small area on the sea would need an infinite number of possible points, the map will be gridded to allow a graph based search.

The grid will be based on the used GRIB file as GRIB files contain their values in a grid structure as well. Furthermore GRIB files come with a definition of which data points are missing (see *2.1 GRIB files*) so a current or wave map can be used to distinguish areas of land and sea as well. Since charts with detailed depth information are expensive and not available in a free format mostly, the algorithm will not take the depth of the sea into account.

The BSH (Nautical Information Service of the Federal Maritime and Hydrographic Agency Rostock, *Nautischer Informationsdienst des Bundesamt für Seeschifffahrt und Hydrographie*) was contacted as part of this project and samples were requested and granted. The format and amount of covered area was not sufficient to support the algorithm significantly. Nevertheless it was a nice experience to get in contact and discuss the project and possibilities to include governmental data in a project like this to provide assistance for non-commercial sailors.

From each field it is possible to drive into one of the eight surrounding fields and for each of these ways the fuel consumption can be calculated. For this calculation the average fuel consumption of the vessel and the correlating speed for this consumption is needed as input from the user.

To calculate the needed fuel, a vector for the intended course of the vessel and a vector of the environmental influences is needed. The course vector results of the speed of the vessel (specified by the user) represented as length of the vector and

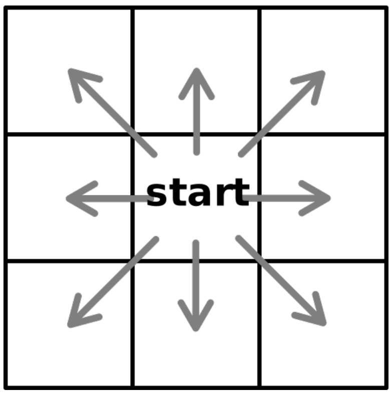


Figure 4.4: A boat can move from each field to the eight surrounding fields possibly in a gridded map

the direction of the course. The environmental vector is represented the same way. Furthermore the distance is needed, which can be extracted from the GRIB file. Each data point in the GRIB has geographical coordinates which can be used to calculate the distance between two points.

With these vectors and the distance the time to cover the distance can be calculated like described in *4.3.3 Vectors and trigonometry*. This time multiplied with the average fuel consumption creates the needed fuel for each grid.

This calculation works well, except if the environmental influences are bigger than the ships velocity, as the ship would go backwards in the search graph.

> *If a current of 7 kn is against a ship which only makes 5 kn, the ship will actually drive backwards or might stay on place if the engine is powered up to gain more speed.*

Even if this heavy current would support the vessels direction, it could go very fast but might become difficult to maneuver regarding fixed points like landmarks, buoys or similar. It is not assumed to be safe to drive into waters with such high currents in this project. As these cases are really rare and are not representable by the proposed algorithm, such fields will be considered as unreachable for simplicity.

To calculate the fuel consumption over longer distances and time, there are different approaches which are discussed in the following sections. The approaches are designed to match the different requirements as stated in *4.1.3 Time vs Quality* and *4.2.4 Various solution patterns.*

4.3.4.1 Predetermined paths

The most simple and straight forward solution is to calculate the fuel consumption of a path specified by a user. In this case a list of coordinates can be processed without any decision trees or comparisons to find the most useful path.

The expense of this solution pattern has a complexity of $O(n)$, with n as the length of the list of coordinates. This solution assumes that the user already has an idea of the best way to go, which may require specific knowledge of the area. On the other hand, this may be a quick solution to check a already planned route.

4.3.4.2 Path between two points

As the most common use case might be to calculate the fuel consumption between two points on the map it would be desirable to provide an algorithm with start, goal and start time as input. A highly suitable algorithm for tasks like this is the A* search, as it has a good runtime, is complete and optimal at the same time, if the heuristic is consistent. The paper mentioned in *3.2 Long-term routing for autonomous sailboats* uses a A* approach as well and achieved good results.

The quality of an A* approach depends mainly on the used heuristic. The heuristic in this case is the fuel consumption over the fastest, direct route, as this would require the least fuel.

The fastest currents are tidal currents if the water has to pass a small passage for example between two islands. One of the fastest currents in the world according to the *Norwegian Environment Agency* is at Saltstraumen and can reach a speed up to 22 kn [Tho16]. But in most areas the strength of currents is commonly in the range of low single digits (0 kn - 4 kn). To calculate with the fastest possible current in perfect angle would make the heuristic to optimistic.

This is why the heuristic will use the speed of the vessel as the highest value for the current, as the algorithm will ignore fields with higher currents anyway. This value is high enough to ensure that no current will exceed this value, but low enough to make the heuristic not too optimistic.

This way the heuristic should be admissible and consistent, as for every node the heuristic calculates a value lower or equal the real costs and will not get less accurate for the next steps.

4.4 Implementation

This chapter explains the implementation of the project. The code is written in python as it is a highly suitable language for mathematical tasks and highly portable to different operation systems and architectures. As the data in GRIB files is structured as sparse arrays, numpy is highly suitable to work with these efficiently. The complete code can be found in appendix *A Source code.*

The code is structured as follows: As can be seen in the directory tree (see *4.6 Directory structure*), in the main directory is the directory *fuelcalculator*, which contains the files of the developed fuel calculator library. In *gribFiles* are some example GRIB files for testing purposes. The folder *output* is used to store results generated by the test scripts. In *static* the web part of the GUI is located. The CSS and JavaScript files with "bootstrap" in their names are from the Bootstrap framework as the GUI is designed with this framework. [boo22]

The Java Script framework JQuery [Fou22] is used to implement the businesses logic of the GUI.

The Flask framework is used to provide a REST API to exchange data between the client (web browser) and the server part of the GUI as shown in *4.5 System overview.*

The *Dockerfile* and the *docker-compose.yml* are used to run the all the code produced in the project with Docker to simplify dependency management and provide portability.

The *test.py* contains the unit tests for the library. *testFuelConsumption.py* is a

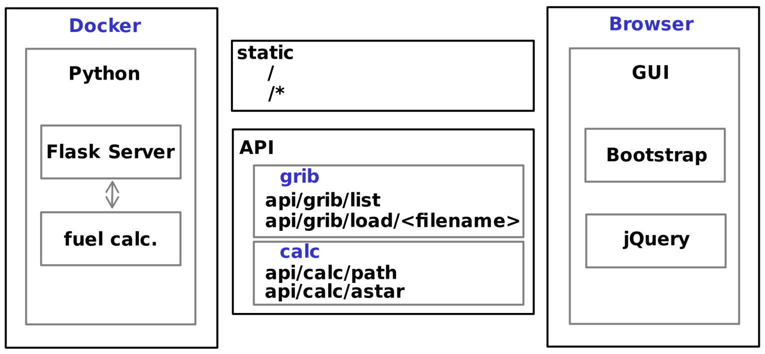


Figure 4.5: The GUI communicates with the server via REST calls. The server runs in a Docker container to abstract from the operation system and provide architectural independency.

test script to test the functionality of the calculation of fuel consumption isolated, by creating test cases with predictable outcomes. The scripts *testPredetermined-Path.py* and *testAstar.py* test the two approaches to calculate the fuel over longer distances and run some runtime tests. The results and findings from these tests are discussed in *6 Validation and evaluation*. The *testArrays.py* contains some helper functions to generate and manipulate numpy tests Arrays for the test cases.

In the coming chapter the implementation *fuel calculator library* will be explained, as the central part of the project. The other test scripts and the GUI are for validation and demonstrating the library only and will therefore not further discussed here. The implementation of the library consists of the handling of GRIB files, the calculation of the fuel and the calculation of paths.

4.4.1 Handling GRIB files

To manage each GRIB file and the data saved in it the class *Grib* will be used. The code for this class can be found in Appendix *A.1.2*. Each Grib object requires a suitable GRIB file, so when initialized several parameters are extracted from the file and checked if and how they are usable, which will be described in detail below.

One of the basic concepts of the project is the usage of GRIB files to obtain

```
fuelcalculator
├── fuelcalculator
│   ├── __init__.py
│   ├── fuelcalculator.py
│   ├── grib.py
│   ├── util.py
│   └── vessel.py
├── gribFiles
├── output
├── static
│   ├── css
│   │   └── bootstrap.min.css
│   ├── js
│   │   ├── bootstrap.bundle.min.js
│   │   ├── jquery-3.6.0.min.js
│   │   └── main.js
│   ├── favicon.ico
│   └── index.html
├── docker-compose.yml
├── Dockerfile
├── README.md
├── server.py
├── test.py
├── testArrays.py
├── testAstar.py
├── testFuelConsumption.py
└── testPredeterminedPath.py
```

Figure 4.6: The file structure of the project

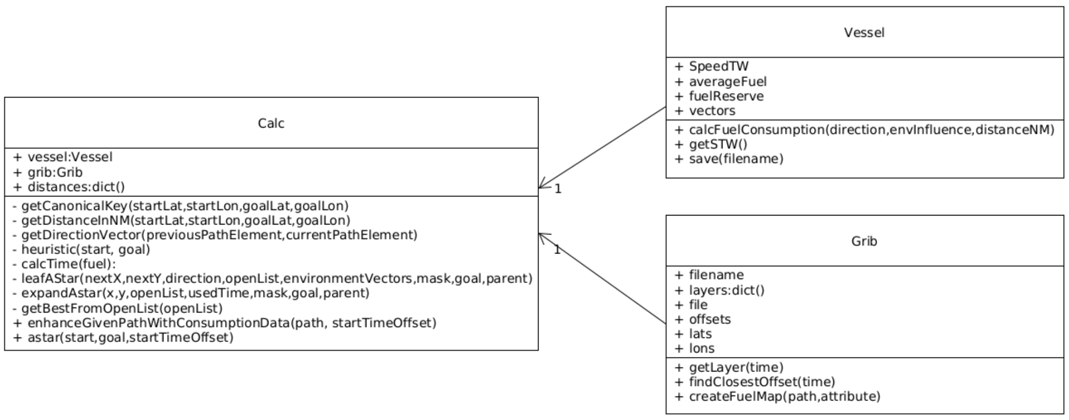

Calc
+ vessel:Vessel
+ grib:Grib
+ distances:dict()
- getCanonicalKey(startLat,startLon,goalLat,goalLon)
- getDistanceInNM(startLat,startLon,goalLat,goalLon)
- getDirectionVector(previousPathElement,currentPathElement)
- heuristic(start, goal)
- calcTime(fuel):
- leafAStar(nextX,nextY,direction,openList,environmentVectors,mask,goal,parent)
- expandAstar(x,y,openList,usedTime,mask,goal,parent)
- getBestFromOpenList(openList)
+ enhanceGivenPathWithConsumptionData(path, startTimeOffset)
+ astar(start,goal,startTimeOffset)
Vessel
+ SpeedTW
+ averageFuel
+ fuelReserve
+ vectors
+ calcFuelConsumption(direction,envInfluence,distanceNM)
+ getSTW()
+ save(filename)
Grib
+ filename
+ layers:dict()
+ file
+ offsets
+ lats
+ lons
+ getLayer(time)
+ findClosestOffset(time)
+ createFuelMap(path,attribute)
1
1

Figure 4.7: Each *Calc* object requires a *Vessel* and a *Grib* object. *Vessel* contains all parameters of the vessel set by the user and contains the *calcFuel* function. *Grib* is used to process a GRIB file, extract required information and provide functions to extract the GRIB layers. *Calc* contains the functions for “predetermined path” and A*.

environmental data. To make use of the information saved in GRIB files, the python library *pygrib* [Whi22] is used. The library can open a GRIB file as a list of dictionaries and provides functions to reach needed information more easily.

Each element of the list represents one time slot and contains all needed information as a dictionary, in which the keys match the names as defined in the code tables of the GRIB2 standard. With this keys, parameters can be looked up easily.

Right after the code opened the provided GRIB file, the usability of the data is checked. To accomplish this, the codes for *discipline*, *parameter category* and *number* are looked up to check the type of data (see *2.1 GRIB files*). If *discipline* is 10 and *parameter category* is 1, the data describe currents in either eastward or northward direction. As the implementation focuses on this type only all other types get rejected for now.

```python
class Grib:
    def __init__(self, filename):
        self.filename = filename
        self.layers = dict()

        # open the GRIB file
        self.file = pygrib.open(self.filename)

        # check if Grib file is usable. For now: It contains currents (
            discipline = 10 and parameterCategory = 1)
        if self.file[1]['discipline'] != 10 or self.file[1]['parameterCategory']
             != 1:
            raise Exception("Unsupported GRIB format")
```

Each data point is related to a coordinate on the globe, saved as pair of latitudes and longitudes. To access the coordinates of data points, they are saved in two 2-dimensional lists which match with the list containing the actual data values. This makes it easier to gain the coordinates for data points later.

```
        self.lats, self.lons = self.file[1].latlons()
```

4.4.1.1 Unit of time

As the data is connected to time, the timestamps and their unit of time used in this GRIB file are needed. To be able to find the matching timestamp later, the timestamps are saved in a sorted list under the use of *map*. This way python will use the most efficient way to collect the needed data depending on the operating system as map is implemented 'native' in Python.

```
        # get the a list of layers offsets
        self.offsets = sorted(list(set(map(lambda grib: grib['forecastTime'],
            self.file))))
```

To get the unit of time, the value of *stepUnits* has to be looked up according to GRIB2 code table 4.4 [WMO21]. In this case only the smallest units are covered, as units like *decades* are unlikely for such data. Depending on the unit code the timestamps get customized, so they represent always minutes, as this is the smallest unit. For example a unit code of 1 stands for hours, so each timestamp gets multiplied with 60.

```
        # The unit of time code is saved in 'stepUnits', see GRIB2 Code Table
            4.4
        # https://www.nco.ncep.noaa.gov/pmb/docs/grib2/grib2_doc/grib2_table4-4.
            shtml (visited 07.04.22)
        layer = self.file[1]

        #found most likely a bug in pygrib
        dummy = str(layer) + str("dummy")

        unitTime = layer['stepUnits']

        if unitTime == 1:
            self.offsets = [x * 60 for x in self.offsets]
        elif unitTime == 2:
            self.offsets = [x * 1440 for x in self.offsets]
        elif unitTime == 10:
            self.offsets = [x * 180 for x in self.offsets]
        elif unitTime == 11:
            self.offsets = [x * 360 for x in self.offsets]
        elif unitTime == 12:
            self.offsets = [x * 720 for x in self.offsets]
        else:
            if unitTime != 0:
                raise Exception("Error: unknown time unit")
```

4.4.1.2 Getting the values from the GRIB

The environmental influences need to be in a vector-like format to calculate the fuel consumption later. As there is one value per layer in the GRIB format only, all the data chunks of the same time frame are collected and joined into one element.

To achieve this all elements with the corresponding timestamp are selected and saved as *east* and *north*. If the *parameterNumber* is 2 the data represent the *Eastward sea water velocity* and if it is 3 it represents *Northward sea water velocity*. From these elements the values get extracted and stacked on each other, creating a 3-dimensional list, which can be interpreted as a 2-dimensional array containing vectors.

The created vector array is saved in a dictionary with the timestamp as key. This way each data layer is extracted once and only looked up later to make the code run faster. This way a simple cache is implemented. An impressive speed gain is expected from this and will be evaluated later.

```
    def getLayer(self, time):
        #is it in cache?
        if time not in self.layers:
            # discipline = 10, parameterCategory = 1, parameterNumber = 2 -> Eastward sea water velocity
            # discipline = 10, parameterCategory = 1, parameterNumber = 3 -> Northward sea water velocity
            byTime = self.file.select(forecastTime=time)
            east = list(filter(lambda x: x['parameterNumber'] == 2, byTime))
            north = list(filter(lambda x: x['parameterNumber'] == 3, byTime))
            self.layers[time] = np.dstack([east[0].values, north[0].values])
        return self.layers[time]
```

4.4.1.3 Find the matching offset

To choose the matching GRIB data layer to a corresponding timestamp, this timestamp is needed. To find the best matching timestamp to a given point in time the function *findClosestOffset* searches for the best match in the list of offsets. The function uses the bisect module, which uses a bisection algorithm and is therefore faster than a linear search approach.

As *bisect_left* returns the position where the value would fit in, the following if else statements are made to figure out which of the neighboring values is closer.

```
    def findClosestOffset(self, time):
        pos = bisect_left(self.offsets, time)
        if pos == 0:
            return self.offsets[0]
        if pos == len(self.offsets):
            return self.offsets[-1]
        before = self.offsets[pos - 1]
        after = self.offsets[pos]
        if (after - time) < (time - before):
            return after
        else:
            return before
```

4.4.2 Configuration of the vessel

The class *Vessel* (see *A.1.4 vessel.py*) aggregates the values of speed, corresponding fuel consumption at this speed and the fuel reserve (tank volume) of the vessel.

As representation of the intended movement of the vessel into the eight cardinal directions vectors "*vectors*" are used. The vectors represent the intended direction and speed (given by the user), which is represented as the length of each vector.

The first entry of the vectors always represent the eastward direction, the second the northward direction to match the vectors of the GRIB layers. So the vector into northern direction has a *0* for east and the speed of the vessel for north. To gain a vector which has still the length representing the speed of the vessel, but point for example in the northeastern direction, the values for *east* and *north* are $\frac{speed}{\sqrt{2}}$ as the length of a vector is $|\overrightarrow{v}| = \sqrt{x^2 + y^2}$ and x and y are the same in this case.

```python
class Vessel:
    def __init__(self, speedTW, averageFuel, fuelReserve):
        self.config = {
            "speedTW": speedTW,
            "averageFuel": averageFuel,
            "fuelReserve": fuelReserve
        }
        # The vessel vectors result from the speed of the vessel.
        # Each vector represent the direction and speed of the vessel into 8
            cardinal points
        self.vectors = {
            "N": [0, speedTW],
            "NE": [speedTW / np.sqrt(2), speedTW / np.sqrt(2)],
            "E": [speedTW, 0],
            "SE": [speedTW / np.sqrt(2), -speedTW / np.sqrt(2)],
            "S": [0, -speedTW],
            "SW": [-speedTW / np.sqrt(2), -speedTW / np.sqrt(2)],
            "W": [-speedTW, 0],
            "NW": [-speedTW / np.sqrt(2), speedTW / np.sqrt(2)]
        }
```

To be able to store the configuration parameters of the vessel, the function *save* uses the python library *pickle*. To create a vessel object from data saved to the disk the function *createVesselFromConfig* is used, which loads a stored config.

```python
    def save(self, fileName):
        try:
            with open(fileName, 'wb') as f:
                pickle.dump(self.config, f)
        except IOError:
            print("Error safe config")
```

4.4.3 Calculate the fuel consumption

The calculation of the fuel consumption works as described in *4.3.3 Vectors and trigonometry*. As it relies on class parameters of *Vessel*, this function is part of the vessel class. The function requires a *direction*, which is the vector representing the intended speed and direction of the vessel, the *envInfluence* which is the vector of the environmental influences and the distnace (*distanceNM*) which will be covered.

First the function checks if the environmental influences are bigger or equal to the vessels speed. In this case the function returns *infinity* to mark this scenario as unreachable. If not, the function calculates the needed fuel.

```python
    def calcFuelConsumption(self, direction, envInfluence, distanceNM):
        #is the environmental influence stronger than the "moving force" of the
            vessel
        if np.linalg.norm(envInfluence) >= self.config["speedTW"]:
            #mark as unreachable
            return float('inf')
```

The first step in the fuel calculation is to calculate the angle between the direction vector and the environmental influence. To do this, a function of util (see Appendix *A.1.3* util.py) is used. The util.py file contains several helper functions which calculate angles or sides of triangles like described in *4.3.3 Vectors and trigonometry*. In this case *angleOfEnvInfluence* is called which calls *getAngle*. The function calculates first the length of the given vectors and checks if one of them is close to zero, this prevents a division by zero. In this case no angle can be created so *nan* is returned.

With the length of the vectors the vectors get normalized and the dot product (scalar) is calculated with *numpy.dot*. From the result the arccos gets calculated which then has to be transformed into degrees as numpy functions return angles in radians.

```python
def getAngle(v1, v2):
    lengthV1 = np.linalg.norm(v1)
    lengthV2 = np.linalg.norm(v2)

    isV1Small = np.isclose(lengthV1, 0)
    isV2Small = np.isclose(lengthV2, 0)
    if isV1Small or isV2Small:
        return np.nan

    unitV1 = v1 / lengthV1
    unitV2 = v2 / lengthV2
    dot_product = np.dot(unitV1, unitV2)
    rad = np.arccos(dot_product)
    deg = np.rad2deg(rad)
    return deg
```

Depending on the value of *influenceAngle* the calculation of the fuel consumption differs. If the angle is neither near 0°, 180° nor *nan*, the fuel consumption can be calculated with the "triangle technique", else the vectors are added as described below.

To calculate the angles of the triangle, the velocity of the influence is needed as this represents one side of the triangle. With this velocity, the speed and the influence angle, the second angle (course correction) gets calculated and next the last angle. With all three angles known the DTW can be calculated by the use of the DOG as one side of the triangle. From this the time and then the fuel is calculated.

```
        influenceAngle = util.angleOfEnvInfluence(direction, envInfluence)
        if 0.1 < influenceAngle < 179.9 and not np.isnan(influenceAngle):
            stw = self.config["speedTW"]
            influenceVelocity = util.velocityOfEnvInfluence(envInfluence)
            cca = util.courseCorrectionAngle(stw, influenceAngle,
                influenceVelocity)
            angle = util.getThirdAngleOfTriangle(influenceAngle, cca)
            dog = distanceNM
            dtw = util.getDTW(angle, influenceAngle, dog)
            ttw = util.getTTW(dtw, stw)
            avgConsumption = self.config["averageFuel"]
            return util.getFuelConsumption(ttw, avgConsumption)
```

If force of the influence is nearly nonexistent (near zero) or the influence and the moving direction work in the same direction (positive or negative), the vectors are added and the length of the resulting vector represents the SOG. From this speed the *time through water* (the time to reach the desired point) gets calculated. The fuel consumption for this way is calculated afterwards.

```
        else:
            cog = np.add(direction, envInfluence)
            sog = np.linalg.norm(cog)
            ttw = util.getTTW(distanceNM, sog)
            avgConsumption = self.config["averageFuel"]
            return util.getFuelConsumption(ttw, avgConsumption)
```

4.4.4 Calculate distances

At several points in the code the geographical distance between two points on the globe is needed. The function *__getDistanceInNM* returns the needed distance in the most time efficient and accurate way in NM. As this function requires neither vessel nor grib parameters, this function is part of the class *Calc* in *A.1.1 fuelcalculator.py*.

As in many cases the exact same distance is needed multiple times while calculation, the information gets cached in a dictionary with a canonical key, which is put together from the start and goal latitudes and longitudes. The form is always *smaller longitude, bigger longitude, smaller latitude, bigger latitude*. The fact that the keys are commutative is exploited and the resulting key is used in a hashmap like manner.

For an accurate calculation of the distance the library *geopy* is used. Geopy uses the *WGS-84 ellipsoid*, as it is globally the most accurate ellipsoid representing the shape of earth according to the authors of the library [geo18]. Also the library converts the result into the needed NM.

```
    def __getDistanceInNM(self, startLat, startLon, goalLat, goalLon):
        key = self.__getCanonicalKey(startLat, startLon, goalLat, goalLon)
        # check cache for the key
        if key not in self.distances:
            self.distances[key] = dst.distance(
                (startLat, startLon), (goalLat, goalLon)).nautical
        return self.distances[key]
```

4.4.5 Calculate predetermined paths

The function *enhanceGivenPathWithConsumptionData* of *Calc* is used to calculate the fuel consumption along a given path. As input the function needs the path as a sorted list of coordinates in the form like the values of the GRIB file are and a start time offset. By comparison of the current and the next position of the path the direction for the next step is determined. The consumption is calculated by traversing the path element by element. Each element of the path is enhanced by adding the time, covered distance, the consumed fuel and the leftover fuel to reach the given point.

```python
    def enhanceGivenPathWithConsumptionData(self, path, startTimeOffset):
        mask = self.grib.file[1].values.mask
        previousPathElement = path[0]
        travelTime = 0
        travelDistance = 0
        travelFuel = 0
        enhancedPath = [
            {
                "position": path[0],
                "time": 0,
                "distance": 0,
                "fuel": 0,
                "remainingFuel":  self.vessel.config["fuelReserve"]
            }
        ]
```

The path is processed until it reaches its end or a masked position (no further calculation possible). The data of each position is matched to the time it is 'reached'. The corresponding map (GRIB layer) is used in the calculation.

> *If a user starts at the harbor of Oostende BE at 04:42 UTC towards Calais FR, the current near Dunkerque FR is strong and is directed against the course. At the time the vessel reaches Dunkerque FR at 10:19 UTC the current would have changed and supports the course now. The course planning needs to reflect the currents with their strength and direction at the times the course crosses the position.*

It is intentional that the "*remainingFuel*" can get negative. If the ship has too little fuel on board, it can be seen from the result how much more would be needed.

```python
        # calculate the fuel consumption for every position in path regarding
            the vessel
        for currentPathElement in path[1:]:
            curX, curY = currentPathElement
            prevX, prevY = previousPathElement

            # if the current field is masked, the path is invalid from this
                point and the function ends
            if mask[curX][curY]:
                return enhancedPath

            timeIndex = self.grib.findClosestOffset(travelTime + startTimeOffset
                )
            direction = self.__getDirectionVector(previousPathElement,
                currentPathElement)

            prevLat = self.grib.lats[prevX][prevY]
            prevLon = self.grib.lons[prevX][prevY]
            curLat = self.grib.lats[curX][curY]
            curLon = self.grib.lons[curX][curY]

            distance = self.__getDistanceInNM(prevLat, prevLon, curLat, curLon)
            gribLayer = self.grib.getLayer(timeIndex)

            fuel = self.vessel.calcFuelConsumption(direction, gribLayer[curX][
                curY], distance)
            travelFuel = travelFuel + fuel
            travelTime = travelTime + self.__calcTime(fuel)
            travelDistance = travelDistance + distance
            enhancedPath.append({
                "position": currentPathElement,
                "time": travelTime,
                "distance": travelDistance,
```

4.4.6 Find a way between two points with A*

The code for the A* algorithm is divided into several functions to make it modular. The algorithm is implemented in the function *__astar*, which picks the next candidate. The function *__expandAstar* calls *leafAStar* for each cardinal direction, which then process the new candidates. All functions are part of the class *Calc*.

4.4.6.1 A*

The *astar* function needs the start and goal position and a start time. The function creates an empty *fuelMap* in the same shape of the GRIB file as a masked array. This fuel map is used to visualize the output and will be returned by the function. Right at the beginning, the function also tests whether the start or end point is masked to prevent the algorithm from searching for an unreachable goal.

```
    def astar(self, start, goal, startTimeOffset):
        # initialize the fuel Map filled with zeros
        firstGribLayer = self.grib.file[1].values
        mask = firstGribLayer.mask
        #init empty map
        fuelMap = ma.masked_array(np.zeros((len(firstGribLayer), len(
            firstGribLayer[0]))), dtype=float), mask=mask)
        # open and closed List as Sets to avoid duplicates
        if mask[start[0]][start[1]] or mask[goal[0]][goal[1]]:
            return {"fuelMap": fuelMap, "nominee": None}
        openList = list()
        closedList = set()
        # each tupel: (latitude, longitude, used fuel, usedFuel + heuristic,
            time, parent, distance)
        # add start point to openList
        first = {
            "x": start[0],
            "y": start[1],
            "usedFuel" : 0,
            "validation": self.__heuristic(start, goal),
            "travelTime": startTimeOffset,
            "parent": None,
            "distance": 0
        }
        openList.append(first)
```

The function processes the open list in a loop until the goal is reached or no solution was found (open list empty).

The open list is processed by picking and removing the best candidate with the lowest costs (until now) plus the costs to reach the goal regarding the heuristic.

```
        # loop until goal is reached or no solution possible
        while True:
            # if openList is empty, no solution was found
            if len(openList) == 0:
                return {"fuelMap": fuelMap, "nominee": None}

            #get next
            nominee = self.__getBestFromOpenList(openList)
```

If this candidate is not already in the closed list (there is a better way to reach this point), it will be expanded (its neighbors are put to the open list). The costs are stored to the *fuelMap* array.

The fuel map is build this way and contains a overview of the fuel needed to reach a certain point in best case.

```
            # expand
            if (nominee["x"], nominee["y"]) not in closedList:
```

```python
                curX = nominee["x"]
                curY = nominee["y"]
                usedFuel = nominee["usedFuel"]
                usedTime = nominee["travelTime"]

                # add consumption to the fuel map
                fuelMap[curX][curY] = usedFuel

                # add node to closedList
                closedList.add((curX, curY))

                # check if goal reached
                if curX == goal[0] and curY == goal[1]:
                    return {"fuelMap": fuelMap, "nominee": nominee}

                self.__expandAstar(curX, curY, openList, usedTime, mask, goal, nominee)
```

4.4.6.2 Expansion candidates

The __*expandAstar* function finds for each candidate all possible neighbors and passes them to __*leafAStar* for processing.

```python
    def __expandAstar(self, x, y, openList, usedTime, mask, goal, parent):
        neighbors = [
            [x - 1, y, self.vessel.vectors["N"]],
            [x - 1, y + 1, self.vessel.vectors["NE"]],
            [x, y + 1, self.vessel.vectors["E"]],
            [x + 1, y + 1, self.vessel.vectors["SE"]],
            [x + 1, y, self.vessel.vectors["S"]],
            [x + 1, y - 1, self.vessel.vectors["SW"]],
            [x, y - 1, self.vessel.vectors["W"]],
            [x - 1, y - 1, self.vessel.vectors["NW"]]
        ]
        environmentVectors = self.grib.getLayer(self.grib.findClosestOffset(usedTime))
        for element in neighbors:
            x, y, direction = element
            self.__leafAStar(x, y, direction, openList, environmentVectors, mask, goal, parent)
```

4.4.6.3 Process candidates

The function __*leafAStar* handles each neighbor by checking it for validity. A neighbor is valid if it is reachable (not masked, in bounds of the map) and the environmental influence is not to strong (e.g. the current is not faster than the vessels speed). The function calculates the fuel consumption for each valid neighbor and adds them to the open list.

```python
    def __leafAStar(self, nextX, nextY, direction, openList, environmentVectors,
         mask, goal, parent):
        # expands one leaf and adds it to the open List for A* algorithm
        curX = parent["x"]
        curY = parent["y"]
        usedFuel = parent["usedFuel"]
        usedTime = parent["travelTime"]
        coveredDistance = parent["distance"]

        #is in bounds and not masked?
        if len(environmentVectors) > nextX >= 0 and len(environmentVectors[0]) >
             nextY >= 0 and mask[nextX][nextY] == False:
            curLat = self.grib.lats[curX][curY]
            curLon = self.grib.lons[curX][curY]
            nextLat = self.grib.lats[nextX][nextY]
            nextLon = self.grib.lons[nextX][nextY]

            distance = self.__getDistanceInNM(curLat, curLon, nextLat, nextLon)
            fuel = self.vessel.calcFuelConsumption(direction, environmentVectors
                [curX][curY], distance)
            time = self.__calcTime(fuel)
            currentUsedFuel = usedFuel + fuel
            if fuel != float('inf'):
                newCandidate = {
                    "x": nextX,
                    "y": nextY,
                    "usedFuel" : currentUsedFuel,
                    "validation": currentUsedFuel + self.__heuristic((nextX,
                        nextY), goal),
                    "travelTime": usedTime + time,
                    "parent": parent,
                    "distance": coveredDistance + distance
                }
                openList.append(newCandidate)
```

4.4.6.4 Heuristic

The heuristic needs to be admissible and consistent. The heuristic calculates the best case to reach the goal. This would mean to go the shortest path to the goal with perfect support of the environmental influences (e.g. strongest current as support all the way).

```
    # calculate the heuristic function by multipling the distance to the goal
        with the minimal consumption of the vessel
    def __heuristic(self, start, goal):
        startX, startY = start
        goalX, goalY = goal

        speedTW = self.vessel.config["speedTW"]
        fuelConsumption = self.vessel.config["averageFuel"]

        startLat = self.grib.lats[startX][startY]
        startLon = self.grib.lons[startX][startY]
        goalLat = self.grib.lats[goalX][goalY]
        goalLon = self.grib.lons[goalX][goalY]

        distance = self.__getDistanceInNM(startLat, startLon, goalLat, goalLon)

        return (distance/(speedTW * 2)) * fuelConsumption
```

Chapter 5

Improvement

This chapter is about a possible improvement of the algorithm. In *5.1 Analysis* the limits of the current implementation are pointed out. *5.2 Realization* describes how the algorithm can be improved and in the section *5.3 Use Cases* shows a collection of possible use cases with these improvements.

5.1 Analysis

The A* algorithm as it is implemented works very well for its purpose. It calculates the needed fuel for a path between two points with an acceptable amount of memory and time. It takes into account which way points are reachable and which are not as it distinguishes between land and sea. It finds the optimal solution for its use case.

But as the A* algorithm is basically a algorithm which finds optimal paths, it could solve multidimensional problems as well by finding multidimensional paths. The current implementation is only capable of finding a two-dimensional solution between two points on a geographical map. The goal is to calculate the fuel consumption on this special path. But if the algorithm could take more aspects into account or adjust the preferences, it could solve much more problems.

5.2 Realization

To gain an algorithm capable of solving multidimensional problems a more universal solution is needed. The following sections explain how this could be achieved.

5.2.1 Concept

The A* algorithm finds the best path by always choosing the most promising expansion candidate out of a list. The evaluation of each candidate is $f(n) = g(n) + h(n)$, with $g(n)$ are the costs to reach this point and $h(n)$ are the estimated costs to reach the goal from this point [SN16]. So to gain a universal implementation the function of the heuristic $h(n)$ and the function of the costs $g(n)$ are required to be passed into the function.

Another task of a A* algorithm is to find for each candidate its neighbors. To gain full independence for the algorithm the selection of candidates should also be provided by another function. This way additional considerations can be taken into account, like “which waterways are navigable?”. This might be concerns like the depth of the water or areas which are not open for marine traffic.

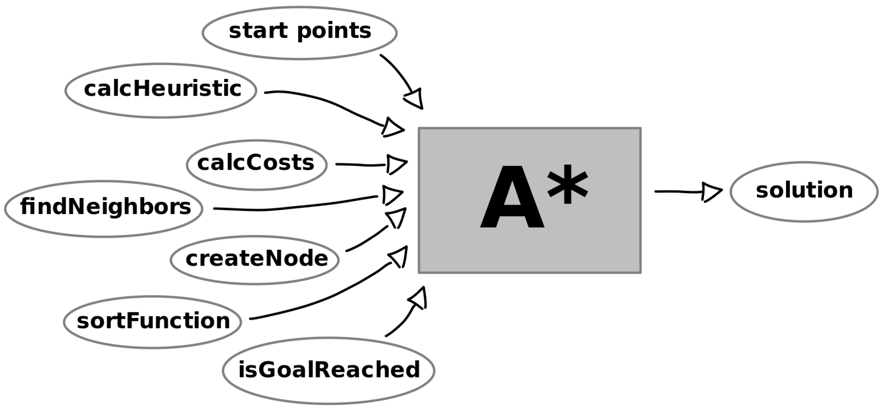


Figure 5.1: The A* gets every decision relevant function and a list of start points as input to make the algorithm universal.

A main requirement of the A* algorithm is a definition of start and goal. To allow multi dimensional solutions, multiple start and end points should be possible. To achieve this the start points can be passed to the function as a list or in another form of collection and to check if the goal is reached another external function could be provided.

By pursuing this concept of outsourcing all major tasks, the A* becomes a universal algorithm which only takes care of calling the individual functions correctly. This way the algorithm can solve multidimensional problems which can be defined easily by implementing the external functions.

5.2.2 Solution

The pseudo code (see *5.2 Pseudo Code for the improved A**) shows an approach to a universal A*. The function gets a list of start points and the external functions.

First the A* creates an open and closed list, adds all start points and extracts the first expansion candidate. After that all candidates are processed in the loop until there are none left or the goal has been reached.

Every candidate has to be checked if it is not already in the closed list. If not, the candidate will be expanded. To do so, first the neighbor function gets called, which returns a list of valid candidates. For all these neighbors the *createNode* function is called, which receives the validation functions *calcHeuristic* and *calcCosts* as well as the current candidate, as each node has to save a reference to its parent to reconstruct the path later with the winner candidate. The calling of the *createNode* function does not necessarily have to happen in a for loop, it could also be realized with *map* in Python for better performance as it is implemented native (build-in function).

As soon as all candidates are added to the open list, it has to be sorted again with the *sortFunction* provided. At this point the candidate is processed, so it is added to the closed list and the next candidate is taken from the list. As the open list is always sorted, the first candidate is always the most promising.

```
function UNIVERSALASTAR(startPoints, calcHeuristic, calcCosts,
        findNeighbors, createNode, sortFuntion, isGoalReached)
    create open list
    create closed list
    openList.addAll(startPoints)
    candidate = openList.getFirst
    while cadidate $\neq$ None or ISGOALREACHED(candidate) do
        if candidate not in closedList then
            neighbors = FINDNEIGHBORS(candidate)
            for all neighbors do
                openList.add(CREATENODE(calcHeuristic, calcCosts,
                    candidate))
            end for
            openList.sort(SORTFUNCTION)
            closedList.add(candidate)
            candidate = openList.getFirst
        end if
    end while
    return candidate
end function
```

Figure 5.2: Pseudo Code for the improved A*

5.3 Use Cases

With the universal approach a variety of use cases opens up. With an algorithm which is capable of multi dimensional solutions many problems can be solved with the same algorithm. Some scenarios are outlined in the next sections.

5.3.1 Multiple starting times

The first additional dimension people think of is time. In this case, it would be the question "which is the best time to start?". In order to answer this, the algorithm can start with several start times. To do so only a list of starting points is needed, which all contain the same geographic start and goal position but different start times.

As all these start points would have the same evaluation in the beginning, the algorithm would start to expand each of them after another. But after some while some (time) paths would have better evaluations than other, as the fuel consumption depends on the environmental influence (e.g. current) at the time.

Now the first path reaching the goal would not only contain the minimal fuel consumption for this way, but also the best time to start.

5.3.2 Solar energy for electrical engines

Over the climate change combustion engines are losing popularity while electric engines are becoming more common, at least for land vehicles. It is most likely that this development will also occur in the maritime sector at some time.

To charge batteries on vessels, solar panels and little wind turbines are popular among many sailors. An interesting approach could be to use a weather GRIB, which describes the occurrence of clouds to predict at which time the most energy can be used directly from a solar panel to power the engine and therefore save battery power and avoid conversion losses.

To do so a heuristic functions and a cost function have to be implemented, which then are passed to the A* function. The functions could offset the expected energy gain from the solar panel depending on clouds and the time of day against the energy needed for the motor depending on the current.

The algorithm would find a way in this case, on which the most energy is left in the batteries, not only taking the consumed energy into account, but also the harvested energy.

5.3.3 Multiple factors

As could already be guessed from the previous chapters the universal implementation of the A* covers many use cases which can contain multiple considerations to reduce the fuel or electric energy for a trip.

But even more evaluation criteria are imaginable. Depending on the preferences of the user the algorithm could evaluate beside the fuel consumption the needed time to reach a goal by adjusting the passed heuristic and cost functions. A user could even choose a path with less waves or winds to have a more pleasant journey by adding and increasing the costs of wind and waves.

Another relevant aspect to plan a route might be fixed times to pass particular points of interest. Many bridges and locks have opening times or a special boat might fit under a bridge with low tide, but not with high tide.

The longer individual journeys become, the more things have to be taken into account. A person which sails a boat all alone has a maximum time to stay awake. A cost function could be implemented, which has after a certain time an exponential growth, representing a person getting more and more tired after 12 hours and will be asleep definitive after 24+ hours regardless of the situation.

Another interesting calculation could be how much drinking water will be needed for a trip, depending on the weather. Of course the longer the journey takes the more water will be needed, but also the hotter and muggy the weather is, the more water each person on board will drink.

These are only a few examples of use cases which all could be solved by a universal implementation of the A* algorithm like described in *5.2 Realization.*

Chapter 6

Validation and evaluation

The goal of the project was to follow the question “How can algorithmically the fuel consumption of pleasure vessels be determined, depending on environmental influences?”. Although there is still room for improvement, the basic concepts show that it is possible to calculate the needed fuel in suitable time ranges without heavy server farms, as will be shown in the following chapter.

The usage of the language Python for this project was a good choice, as the language is highly suitable to handle data and solve mathematical tasks. The library numpy is a welcome addition to handle arrays, as those occurred quite often in this project. Furthermore Python makes the code compatible to most platforms and architectures, such as iOS, Linux and also small board computers like the raspberry Pi (ARM64) or laptops (x86). Therefore the language contributes a major part to the concept *4.2.2 Single board computer*, as Python runs on different platforms and uses an interpreter system, which means no special compilation per platform is needed. Even more abstraction is achieved by the use of Docker.

Caching gave major improvements, as it reduced the runtime. The cached data is particularly advantageous when several calculations on the same GRIB file are run. If, for example, the A* was started for different start times one after another, the following calculations were faster, since many of the required parameters, such as the distances between the points, were already available in the cache.

One of the main requirements was to find a universal data input as stated in *4.1.1 Universal data sources* which is available from different sources, especially also on high seas far away from any WiFi or mobile internet. The usage of GRIB files as source of weather information fulfills this requirement, as GRIB files are already really common in the maritime sector and are also available via satellite connections or other methods.

Due to the fact that for the development process no closed sourced projects where used, the project could be released under a Open Source license and therefore the requirement *4.1.6 Open Source* is fulfilled and also an integration into Open Plotter is possible in future work.

6.1 Calculation of the fuel

The calculation of the fuel per gridded field works very well. By using the method to calculate the influence of currents or similar under the use of vectors the result is pretty accurate. As the function retrieves all parameters needed for the calculation, it is universal and can calculate the fuel for every use case. The decision which environment vector is used or how the distance gets calculated is not taken in this function. All this is implemented by other parts of the program and therefore this function is one example for the concept *4.2.5 Separation of concerns.*

The user has to provide a basic set of knowledge in form of the average fuel consumption of the vessel depending on rpm of the engine (or speed), which can be guessed as stated in *4.3.1 Fuel Consumption of a vessel.* Even if the calculation is based on a value which is too high, it still gives a good estimation.

The tests with the test arrays (*A.2.2 testFuelConsumption.py*) show that the fuel consumption gets calculated as expected. The result can be seen in *6.1 Fuel consumption in a test environment.* For this test the needed fuel was calculated per field, if the vessel drives eastwards. The fuel consumption gets higher the more the environmental influence directs against the ship and gets lower the more the influence supports the direction of the ship, as can be seen on the color charts.

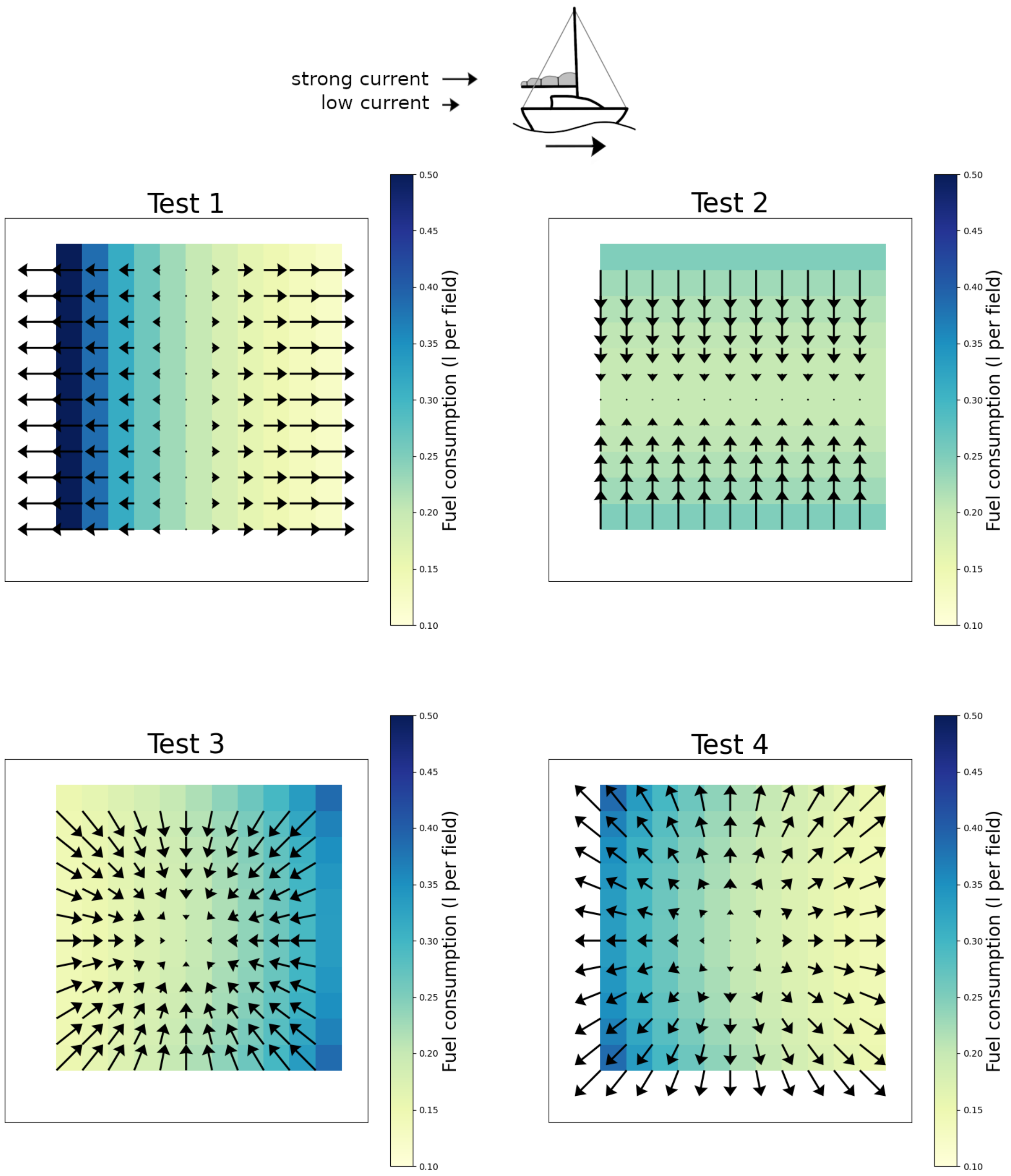
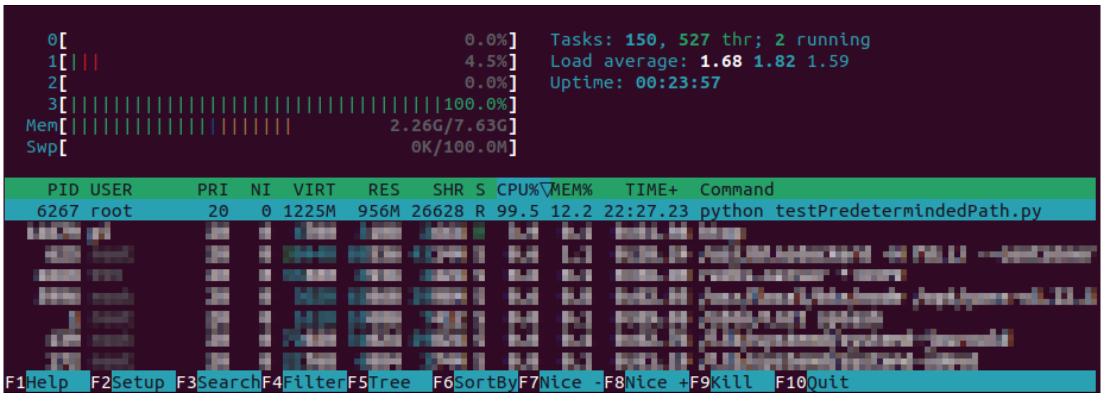


Figure 6.1: The graphs show the fuel consumption in liter per field for a course to the east on each field. One field corresponds to 0.5 NM, the ship consumes 2l per h and moves with 5kn. The arrows represent the environmental influences, varying in their speed from 0 to 3kn. The tests show how the fuel consumption varies depending on the environmental influence.

6.2 Predetermined path

The most basic approach to calculate the fuel consumption, was to calculate it for predetermined paths defined by the user. The goal of this strategy was to get a result quickly to fulfill the "fast result" part of the requirement *4.1.3 Time vs Quality*. The runtime was measured with tests for validation. The tests shown in this section are created with *A.2.3 testPredeterminedPath.py* on a Raspberry Pi 4 (8GB RAM).

The test saturated one CPU core (99.5%) and consumed around 1GB of RAM, as shown in figure *6.2 CPU usage during testing*.



```
    0[                                    0.0%]   Tasks: 150, 527 thr; 2 running
    1[|||                                 4.5%]   Load average: 1.68 1.82 1.59
    2[                                    0.0%]   Uptime: 00:23:57
    3[||||||||||||||||||||||||||||||||||100.0%]
  Mem[||||||||||||||||||||||         2.26G/7.63G]
  Swp[                                0K/100.0M]

    PID USER       PRI  NI  VIRT   RES   SHR S CPU%▽MEM%   TIME+  Command
   6267 root        20   0 1225M  956M 26628 R 99.5 12.2 22:27.23 python testPredetermindedPath.py
[redacted]
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 6.2: The screenshot shows the output of htop during the test of the predetermined Path. The test saturated one CPU core.

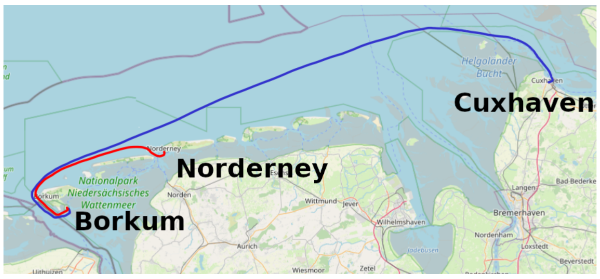


Figure 6.3: The paths used for the test. **red:** Borkum to Norderney, **blue:** Borkum to Cuxhaven

The test cases consist of two routes, from Borkum to Norderney and from Borkum to Cuxhaven. Each route was calculated with two different start times, one at a time with a supporting current (good) and one at a time with a current in the opposite direction (bad). The routes where chosen as they are realistic journeys from harbor to harbor and they differ in their length. The shorter route (Borkum-Norderney) is roughly 28 NM long, the other route (Borkum-Cuxhaven) is roughly 90 NM long.

Even as the test routes are relatively long (if the vessel drives 5 kn at all time it will take 18h for 90 NM) the results are available really fast. So for the use cases in which the user needs a result really fast as in scenarios of “can I reach the harbor” or “which direction should I go” like described in *4.1.3 Time vs Quality*, this is an acceptable solution. Especially since new calculations are much faster, as most of the required data might be already in the cache, which will be described in *6.3 A* approach* later.

route	**start**	**travel time**	**needed fuel**	**runtime**
Borkum - Norderney	good	5h 04 min	10.14l	min: 20.60s max: 22.43s avg.: 21.20s
Borkum - Norderney	bad	5h 31 min	11.04l	min: 22.50s max: 23.51s avg.: 22.84s
Borkum - Cuxhaven	good	17h 36min	35.19l	min: 72.54s max: 78.05s avg.: 74.40s
Borkum - Cuxhaven	bad	17h 55min	35.83l	min: 71.81s max: 78.77s avg.: 73.40s

Figure 6.4: The test includes two predetermined paths (routes), with two different start times each. The good start time was chosen at 2021-11-05 05:40 UTC, the bad start time at 2021-11-05 00:15 UTC. The test was executed on a Raspberry Pi 4. Each test was repeated 20 times to determine an average value. The travel time and needed fuel are the predicted time and fuel the vessel would need for the route. The runtime indicates the minimal, maximal and the average measured run time the calculation needed.

It is noticeable that the shorter routes differ in their fuel consumption significantly more (8.2%) than the longer routes (1.8%). This is because the longer route took nearly 18h and the tide changes roughly every 6h. This means that the vessel has a supporting current at some point and a opposite current at another time, no matter when it starts the journey. The shorter route is less than 6h long, so the difference if the ship goes with or against the current is really noticeable.

The difference of 5% in average fits within the scope of 3-5% which the paper *Ship weather routing: A taxonomy and survey* found as average fuel saving of other researches. This shows that the results found here are consistent with other researches of this topic.

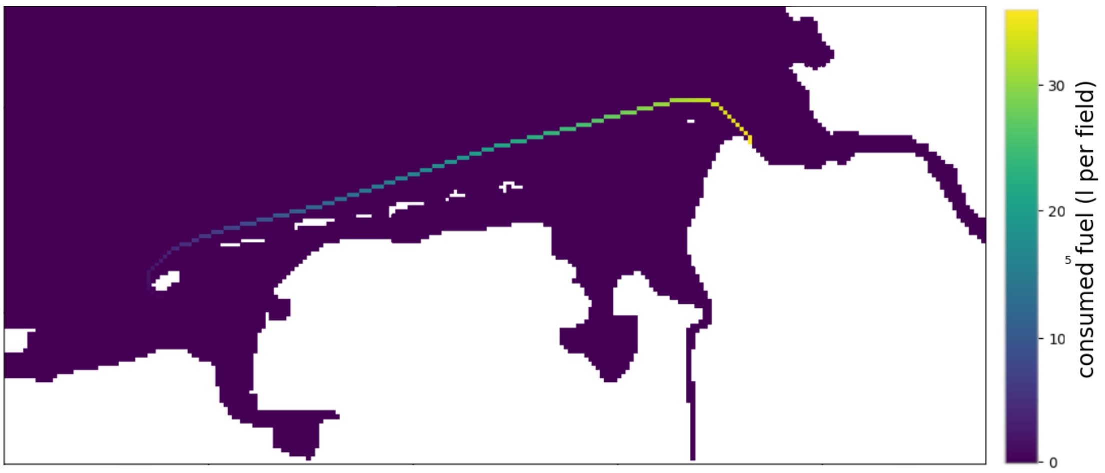


Figure 6.5: The path between Borkum and Cuxhaven. The white area is land, the purple area is sea. The color of the path represents the consumed fuel.

6.3 A* approach

Another way to calculate the fuel consumption is the approach to use the A* algorithm. One of the main concepts is to provide several solution options (see *4.2.4 Various solution patterns*).

The algorithm does not check the needed fuel for a particular path, but finds the path with the lowest fuel consumption between two points. This is both an advantage and a disadvantage, as the algorithm might find more reasonable paths, but does not take enough information into account. The current implementation does not consider water depth, restricted areas or other areas with very special properties (e.g. intertidal areas). The proposed solutions might suggest a path

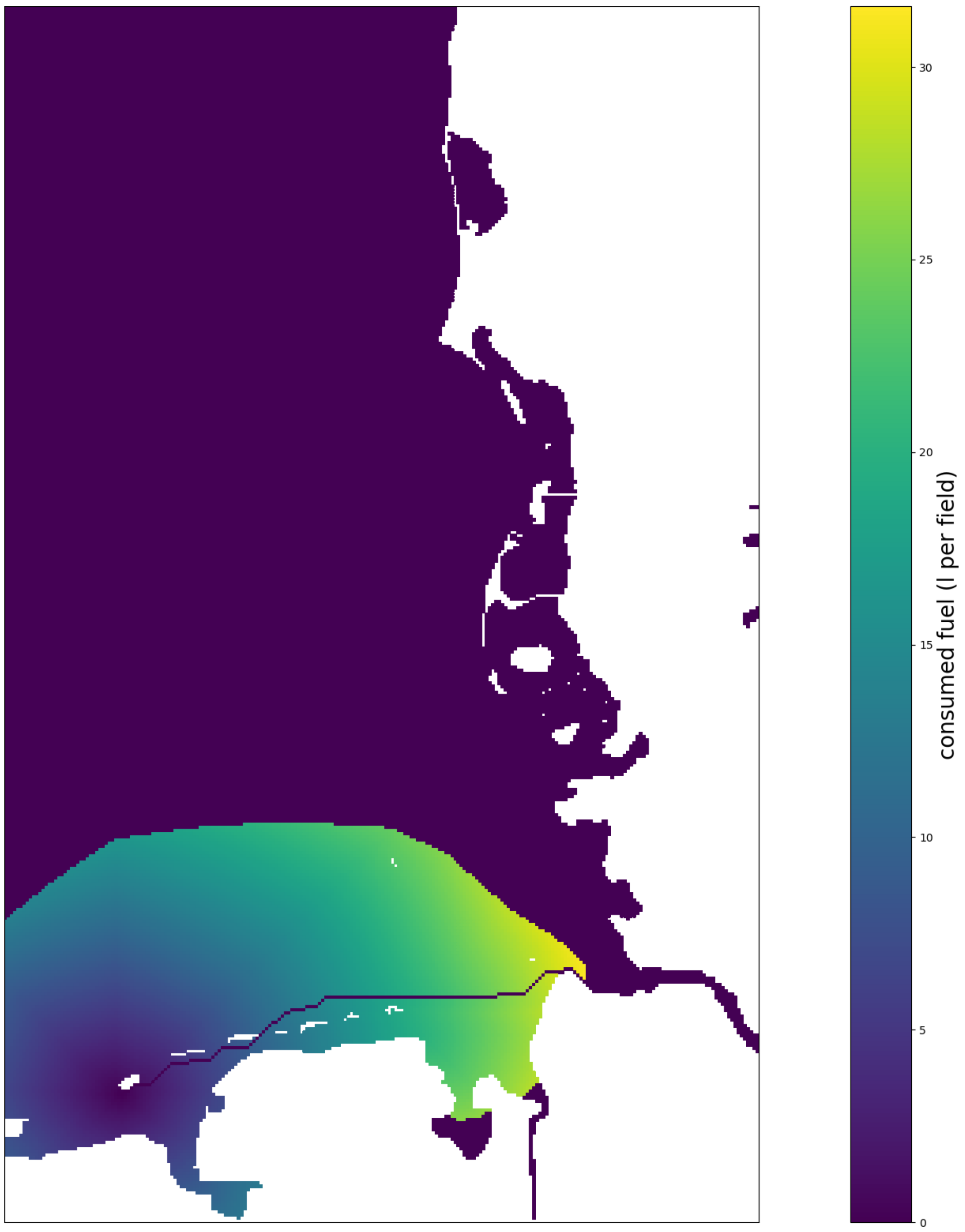


Figure 6.6: The path between Borkum and Cuxhaven, calculated with the A*. The white areas are land, purple is the sea. The colors represent the calculated fuel consumption. The path may be the best solution related to the fuel consumption, but a experienced sailor will notice that this path leads partially through the Wadden Sea, which is not passable for most vessels.

through areas just a few centimeters deep or even out of the water at intertidal mud flats e.g. the "Wattenmeer". A sample output of a route between Borkum and Cuxhaven can be seen at *6.6 A* sample output*.

Despite the problem with low waters, the algorithm finds the paths which are best under the given conditions. An example of this can be seen in *6.7 A* comparison of two routes*, where two results of the same route but with different start times are shown. It is clearly noticeable that not only the paths, but also the area where the A* searched, differ. The left example searched far more into the estuary of the river Ems, while the right example searched more toward the open sea. In the left case the tidal current went toward land (rising tide) and in the right case toward the sea (falling tide), which result into different valuations of expansion candidates.

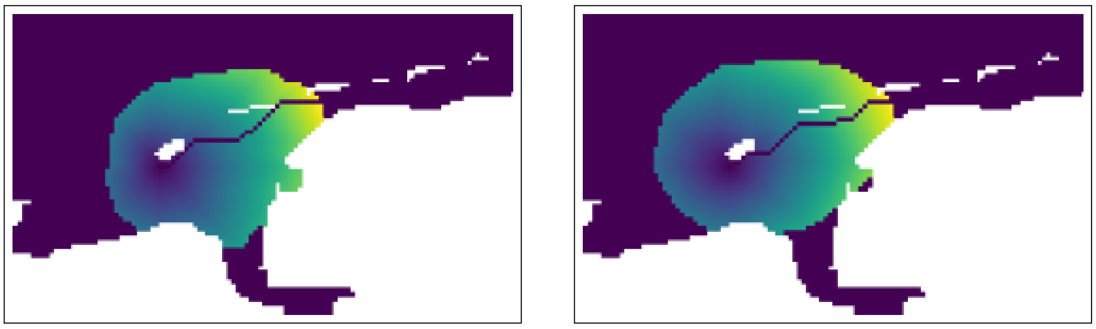


Figure 6.7: The A* calculation for the way between Borkum and Norderney at different start times (rising/falling tide). As can be seen in this comparison, the optimal way varies as it is influenced by the tidal current.

The problem of the required depth of the water could be solved by adding a additional (depth) mask to the map to mark unreachable areas.

To map complex "behavior at sea" could become difficult. Traffic separation areas provide a good example for this. They are basically highways in crowed areas on sea separated for tankers, container ships and alike. Small boats are allowed to cross these areas in a right angle only to interfere as little as possible. Including such considerations would be nice, but is beyond the scope of this project.

Another problem which occurred during testing is named "*sylt error*" in this document as it occurred the first time while calculating a path to Sylt. The island Sylt has a narrow dyke, which cannot be crossed on waterways. As the mask which is used to distinguish land and water is very narrow, the algorithm found a way to slip through diagonally falsely. The result can be seen in *6.8 Bug of A*: The sylt error*.

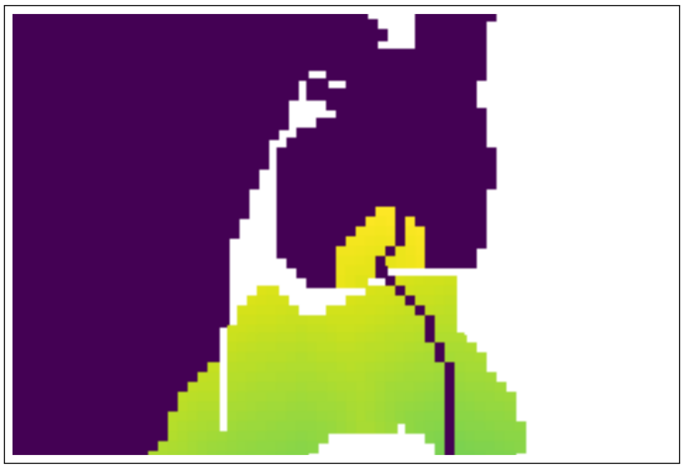


Figure 6.8: The “sylt error”: The image shows a segment of a result of the A* near the island Sylt. The yellow colored fields represent the fuel consumption. The dyke which connects the island Sylt to the mainland is so narrow, that the mask did not prevent the algorithm to go through land. The algorithm found a diagonal way to slip trough, even tough in reality it would not be possible.

To prevent this effect some solutions are imaginable. The mask could be extended by adding around every existing masked pixel 8 masked pixels additionally. Another solution might be to blur the mask and binarize the result by the use of image kernels.

Both these proposals have in common that they should prevent the “sylt error” and close small gaps in the mask. It would expand the coastal lines, but this should not matter as most boats cant drive near the coast anyway. Fine structures like water ways, channels or bays might get corrupted in this process. Additional measures would be needed to avoid this side effects.

The concept mentioned in *4.2.3 Staggered calculations* to base calculations on each other would be applicable to the predetermined path by using the enhanced result (method “enhanced path”) of a previous calculation and further enhancing it. This is not implemented in this work and would need deeper investigation. Therefore the concept is validated as partially fulfilled by design but not implemented as the inclusion of additional environmental inputs became to extensive for the scope of this project.

The usability to calculate the fuel consumption of a route with the A* is fully satisfactory for unexperienced users. It requires a start, goal and start time only.

A test run for a route between Borkum and Norderney (see table *6.9 Runtime test A**) shows: The best time to start the journey is at 2021-11-05T05:00 UTC, as in this case both the time and the fuel are at minimum.

Due to caching of data like distances or layers from the GRIB file the following calculations took half the time of the first calculation in average. This is really good, as most sailors would try different configurations for a trip to see the different outcomes most likely and in this case the caching really saves time.

The test shows a really good usability for inexperienced sailors, as from the results they could conclude to take at least 9 liters of fuel with them to be on the safe side and as an extra the algorithm found the ideal time to start too. This shows that the algorithm can solve the question of “how much fuel will the boat need?” in a easy to use way in a acceptable runtime. With some improvements like a simple layer for water depth, this tool could be even much more useful.

6.4 Concept for improvement

The developed concept to improve the implementation of the A* like described in *5 Improvement* solves several problems of the current implementation. With the universal approach, most of the issues encountered would be easy to solve, whereas with the current implementation extensive refactoring would be necessary for most extensions. Therefore the proposed implementation takes the concept of *4.2.5 Separation of concerns* even more serious, as more of the code is reusable in many contexts. Especially this advantage would make the code really elegant.

start time	**travel time**	**needed fuel**	**runtime**
01:00 UTC	4h 23 min	8.75l	38.04s
02:00 UTC	4h 21 min	8.69l	18.84s
03:00 UTC	4h 15 min	8.51l	18.60s
04:00 UTC	4h 07 min	8.23l	16.34s
05:00 UTC	**4h 04 min**	**8.14l**	**17.89s**
06:00 UTC	4h 07 min	8.23l	17.33s
07:00 UTC	4h 13 min	8.42l	17.85s
08:00 UTC	4h 18 min	8.59l	19.40s
09:00 UTC	4h 22 min	8.72l	18.87s
10:00 UTC	4h 22 min	8.73l	19.07s
11:00 UTC	4h 22 min	8.73l	19.88s
12:00 UTC	4h 20 min	8.67l	19.06s

Figure 6.9: The test calculated the route from Borkum to Norderney each hour for 12 hours to cover a full tide circle. The calculations where done one after another, to benefit from the caches. The start time is the UTC time of 2021-11-05 in this test, the travel time and needed fuel are the predicted time and fuel the vessel will need for the route. The runtime values indicate the measured run time of the algorithm.

Since with the new implementation it will be much easier to adapt the neighboring function, the problems with the missing depth information or the “sylt error” would be much easier to solve. It would require an updated neighbor function only to include additional information about the territory. Fine structures could stay unaffected by this approach.

Furthermore the partly fulfilled concept of *4.2.3 Staggered calculations* could get improved, as with the universal implementation of the A* also the input of environmental influences become universal and can be therefore put together from several sources.

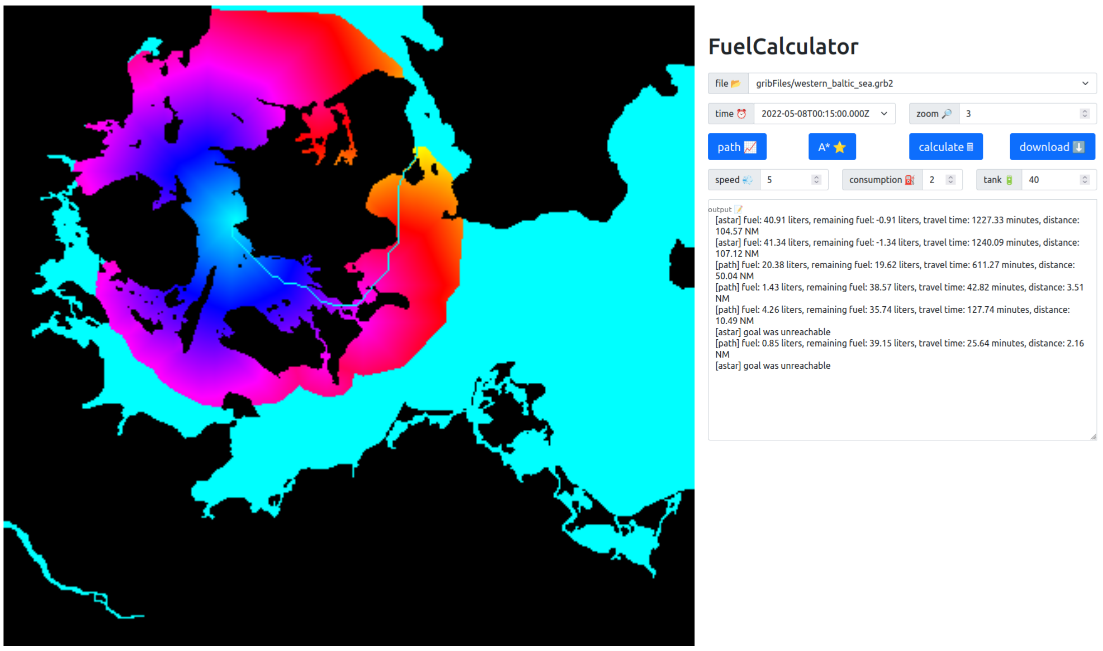
The extensional use cases presented in *5.3 Use Cases* along with the suggested improvements show why the proposed concept should be considered for future work, as they would create many more possibilities and flexibility. The only problem which may occur with the new concept, might be a normalization problem. If both time and fuel consumption get included into the weighting, it might become difficult to compare a time unit with a volume unit. To address this issue would be easier with the suggested improvement too.

6.5 Graphical user interface

The graphical user interface allows easy testing of the algorithms in a descriptive way. It can display various GRIB files and allows the configuration of the parameters. The user can set the configuration of the vessel and choose the start time. For the two approaches of a predetermined path and the search with A* the input can be set by clicking on the map.

The output is illustrated in color on the map and the final results such as needed fuel, leftover fuel and travel time are displayed as text additionally, which fulfills the concept *4.1.4 Output as map.* The GUI presents results in a best effort manner if the input was insufficient or if the calculation did not lead to an entirely complete result. This could happen if a path is drawn partially over land or the goal was not reachable for example (see *6.11 Insufficient input*).

A GPX file can be downloaded which can be imported in other applications. This



file: gribFiles/western_baltic_sea.grb2
time: 2022-05-08T00:15:00.000Z
zoom: 3
speed: 5
consumption: 2
tank: 40

output

```
[astar] fuel: 40.91 liters, remaining fuel: -0.91 liters, travel time: 1227.33 minutes, distance: 
104.57 NM
[astar] fuel: 41.34 liters, remaining fuel: -1.34 liters, travel time: 1240.09 minutes, distance: 
107.12 NM
[path] fuel: 20.38 liters, remaining fuel: 19.62 liters, travel time: 611.27 minutes, distance: 
50.04 NM
[path] fuel: 1.43 liters, remaining fuel: 38.57 liters, travel time: 42.82 minutes, distance: 3.51 
NM
[path] fuel: 4.26 liters, remaining fuel: 35.74 liters, travel time: 127.74 minutes, distance: 
10.49 NM
[astar] goal was unreachable
[path] fuel: 0.85 liters, remaining fuel: 39.15 liters, travel time: 25.64 minutes, distance: 2.16 
NM
[astar] goal was unreachable
```

Figure 6.10: The image shows the GUI with the result of a calculation of a path the path between two points with the lowest fuel consumption. The used GRIB file shows the western Baltic Sea, the numeric outputs of the calculations are printed on the right. In the right upper corner are the selection options for the file and the start time as well as the configuration parameters of the ship.

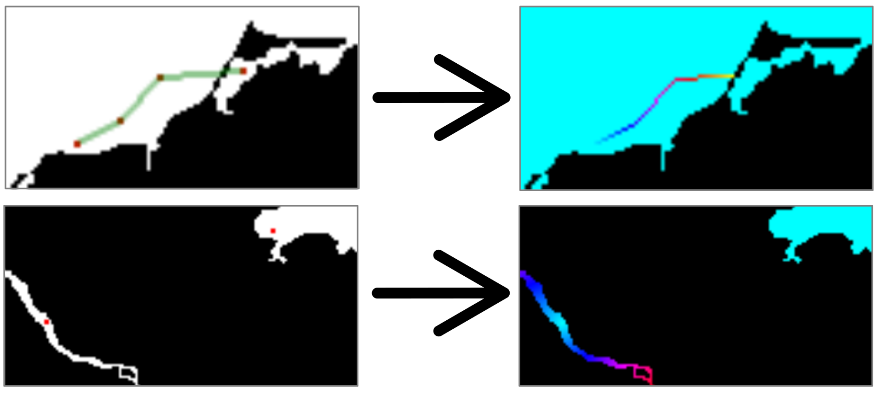


Figure 6.11: If a path is drawn partially over land the result is calculated until a masked spot is reached (upper row). If the goal is not reachable the calculated fuel so far is displayed on the map, but without a path (lower row).

allows the usage of the results with common navigational apps like Navionics as show in *6.12 Path imported in Navionics Boating app*.

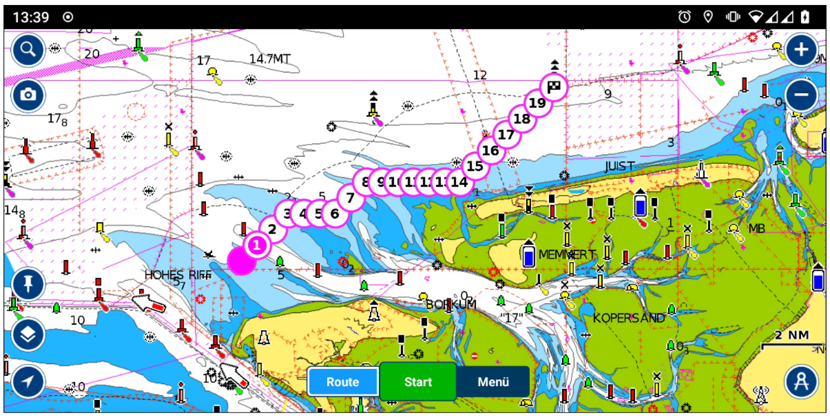
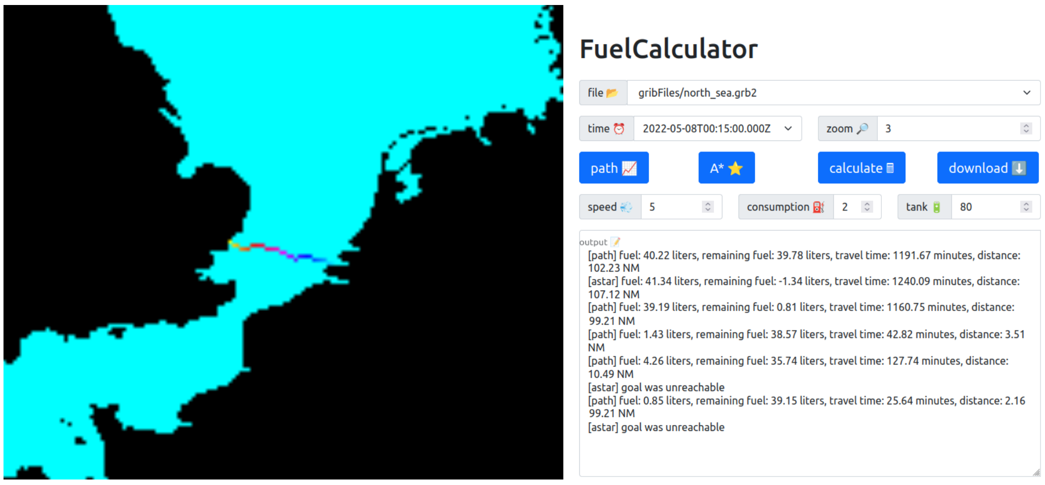


Figure 6.12: A path calculated with A* and imported into the Navionics Boating app (chart plotter).

The trip as described in *1.1 An example use case* from the Netherlands to the UK took about 40 liters diesel per direction. The GUI was used to compare a estimation with the reality. The result of the calculation shows around 40 liters which matches perfectly with consumption experienced, as shown in *6.13 Calculation of the route Netherlands-England*.



FuelCalculator

file 📂 gribFiles/north_sea.grb2

time ⏰ 2022-05-08T00:15:00.000Z

zoom 🔎 3

path 📈 A* ⭐ calculate 🖩 download ⬇️

speed 💨 5

consumption ⛽ 2

tank 🔋 80

output 📝

```
[path] fuel: 40.22 liters, remaining fuel: 39.78 liters, travel time: 1191.67 minutes, distance:
102.23 NM
[astar] fuel: 41.34 liters, remaining fuel: -1.34 liters, travel time: 1240.09 minutes, distance:
107.12 NM
[path] fuel: 39.19 liters, remaining fuel: 0.81 liters, travel time: 1160.75 minutes, distance:
99.21 NM
[path] fuel: 1.43 liters, remaining fuel: 38.57 liters, travel time: 42.82 minutes, distance: 3.51
NM
[path] fuel: 4.26 liters, remaining fuel: 35.74 liters, travel time: 127.74 minutes, distance:
10.49 NM
[astar] goal was unreachable
[path] fuel: 0.85 liters, remaining fuel: 39.15 liters, travel time: 25.64 minutes, distance: 2.16
99.21 NM
[astar] goal was unreachable
```

Figure 6.13: The calculation of the route from the Netherlands to the UK under the use of the path function.

Chapter 7

Conclusion and future work

To measure the success of the project the validation was done in *6 Validation and evaluation* and the different aspects of the concepts and their implementation where weighted against the usability. As the result of the project is a profound implementation of the algorithms with a GUI and a way to import the calculations results in at least one of the most popular commercial chart plotters a fully satisfactory result can be concluded. Minor compromises regarding the concepts were made to keep the project in boundaries and tailor the scope to a sensible scale.

As the calculations run in very sufficient time on hardware suitable for a DC powered vessel environment with a consumption suitable for ship batteries the system proved its practicality to all extends.

The use of GRIB files has turned out to be advantageous as this data format delivers all needed information and is available from multiple sources. With python as programming language, the library pygrib and numpy the GRIB files could be used and interpreted efficiently. The information gained from the GRIB files was successfully integrated into the calculations, not only for the environmental influences, but also as a source of basic maps, time and coordinates.

The method to calculate the fuel like described in *4.3.3 Vectors and trigonometry* is based on a technique long used by many sailors and is therefore in the field for many decades, probably centuries.

The implementation of the predetermined path shows that results can be available really quickly and the A* showed that also without other information a path can be found and calculated in a short time. Beside the good results of the A* approach, during the research improvements where identified, which is great. These improvements could be implemented in future projects. The proposed solutions for the problems identified namely the *Sylt error* or a missing overlay for passable and impassable areas should be considered too.

Furthermore the GUI could get improved for a better usability or a full integration in Open Plotter could be done in a future project especially if more options get developed. Another possible improvement would be a connection to a vessels network, as this would enable even more features, as described below.

7.1 Possible implementations

In the chapter *5 Improvement* a concept was proposed to rewrite the A* algorithm in order to make it more universal. This concept should definitively be implemented in future versions, as it opens up completely new dimensions of possibilities. With the multi-dimensional approach more complex paths could be implemented.

As with the universal approach the heuristic function and cost function are injected to the A* algorithm different evaluation criteria become possible:

- rating by fuel consumption, but in a combination of sailing and motor
- rating by time
- rating by comfort
- rating by waking time
- rating by required reserves like water or food

A evaluation function could be implemented, which tries to find the way with the lowest fuel consumption, but by including path segments which can gone under sail. This could be achieved by using a polar diagram of the vessel and information about the wind. A route, which combines parts under sail with parts driven by engine, could be calculated. The engine would be used if the wind is not sufficient only.

By adjusting the cost function regarding the preferences given by the user the calculation could optimize for fuel, for time or a weighted combination of both.

Another interesting evaluation (cost + heuristic function) could be based on comfort, in the sense of moderate wind and waves combined with a preferable angle to wind and waves as an example. This might be interesting for unexperienced sailors or people prune to sea sickness. There are much more resources on a vessel which can be included in evaluations, like water, food or time a sailor can stay awake to only name a few.

To calculate these evaluations named above several inputs could be used. All weather related topics can be accessed by the usage of different GRIB files, others may come from other sources. Imaginable influences could be:

- current
- wind
- waves
- temperature
- passing times of bridges, locks, etc.
- preferred times to be at certain places

Current and wind as input are probably the most basic parameters, but others are also possible. Waves would be important for the "comford evaluation" and the temperature could be used as a criteria for needed water on a tour.

Fixed times like passing times of bridges or locks could be included into the calculations but also times when the travelers want to be in certain places, to reach a harbor before the grocery store closes in the evening or on longer tours leaving a certain region before the hurricane season or similar.

All these ideas could be added to the implementation together with the improved version of the A*.

7.2 Connection to a vessels network

A connection to the vessels network includes two aspects. First of all it means that a network like WiFi on the vessel exists, which allows different devices to connect to each other in order to exchange information. The more interesting network nevertheless would be the ships network, which shares information from different maritime components and devices. These networks can be build on several standards, a serial bus system like the NMEA 0183 standard, a CAN bus system like NMEA 2000 or even a web standard based format like SignalK.

To connect to such maritime standards, a integration into Open Plotter would be suitable, as it includes already interfaces to common maritime systems. Furthermore it is already in use by many boaters who like to build systems for their own boat and contains several other Open Source projects.

A connection to the vessels network would allow an input to provide the tank level for example. It would even create the possibility to implement an algorithm to provide adjusted consumption data dependent on the rpm of the engine to the current implementation, so a user could see how much fuel could be saved if the boat drives slower.

Bibliography

[Bar90] Axel Bark. *Segelführerschein BR + Sportbootführerschein See.* Delius, Klasing & Co. Bielefeld, 1990.

[BidpemoWM77] Paris Bureau international des poids et measures, International Bureau of Weights, and Measures. *The International System of Units (SI).*, volume 330. US Department of Commerce, National Bureau of Standards, 1977.

[boo22] Bootstrap. `https://getbootstrap.com/`, 2022. visited 12.05.2022.

[DEp21] Sean DEpagnier. Weather routing pi for open cpn. `https://opencpn-manuals.github.io/main/weather_routing/index.html`, 2021. visited 18.05.2022.

[Fou22] OpenJS Foundation. jquery. `https://jquery.com/`, 2022. visited 16.05.2022.

[geo18] Geopy. `https://geopy.readthedocs.io/en/stable/#module-geopy.distance`, 2018. visited 18.05.2022.

[gri03] Guide to the wmo table driven code form used for the representation and exchange of regularly spaced data in binary form: Fm 92 grib edition 2. Technical report, WMO, 2003.

[HK70] Michael Held and Richard M Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[kvr09] Internationale regeln von 1972 zur verhütung von zusammenstößen auf see (kollisionsverhütungsregeln - kvr) teil a, regel 3. `https://www.gesetze-im-internet.de/seestro_1972/BJNR008160977.html#BJNR008160977BJNG000100328`, 2009. visited 18.05.2022.

[LSF11] Johannes Langbein, Roland Stelzer, and Thom Frühwirth. A rule-based approach to long-term routing for autonomous sailboats. In *Robotic sailing*, pages 195–204. Springer, 2011.

[Ltd19] Raspberry Pi (Trading) Ltd. Raspberry pi 4 model b datasheet. `https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf`, 2019. visited 15.05.2022.

[Off10] Met Office. Fact sheet number 6: the beaufort scale. 2010.

[Ope21] OpenPlotter. Openplotter. `https://openmarine.net/openplotter`, 2021. visited 17.03.2022.

[Pen16] AB Volvo Penta. Volvo penta diesel d1-13, 2016.

[sai21] Sailgrib. `https://www.sailgrib.com/`, 2021. visited 03.05.2022.

[Sle19] Steve Sleight. *Segeln: das neue Praxishandbuch*. Dorling Kindersley, 2019.

[SN16] Russell Stuart and Peter Norvig. *Artificial intelligence: A modern approach (Global edition)*. 2016.

[Tho16] Markus Thonhaugen. Er saltstraumen egentlig verdens sterkeste tidevannsstrøm? (english: Is saltstraumen really the world's strongest tidal current?). `https://www.nrk.no/nordland/er-saltstraumen-egentlig-verdens-sterkeste-tidevannsstrom_-1.12929482`, 2016. visited 22.04.2022.

[Whi22] Jeff Whitaker. pygrib. `https://github.com/jswhit/pygrib`, 2022. visited 21.04.2022.

[WMO21] WMO. Manual on codes, volume i.2 (wmo-no. 306), 2021.

[ZPD20] Thalis PV Zis, Harilaos N Psaraftis, and Li Ding. Ship weather routing: A taxonomy and survey. *Ocean Engineering*, 213:107697, 2020.

Appendix A

Source code

This appendix contains the complete code used for this bachelor thesis.

A.1 fuelcalculator

A.1.1 fuelcalculator.py

```python
from geopy import distance as dst
import numpy.ma as ma
import numpy as np

class Calc:
    def __init__(self, vessel, grib):
        self.vessel = vessel
        self.grib = grib
        # cache of distances
        self.distances = dict()

    def __getCanonicalKey(self, startLat, startLon, goalLat, goalLon):
        # find the commutative key
        if startLat < goalLat:
            if startLon < goalLon:
                return str(startLat) + str(goalLat) + \
                    str(startLon) + str(goalLon)
            else:
                return str(startLat) + str(goalLat) + \
                    str(goalLon) + str(startLon)
        else:
            if startLon < goalLon:
                return str(goalLat) + str(startLat) + \
                    str(startLon) + str(goalLon)
            else:
                return str(goalLat) + str(startLat) + \
                    str(goalLon) + str(startLon)

    def __getDistanceInNM(self, startLat, startLon, goalLat, goalLon):
        key = self.__getCanonicalKey(startLat, startLon, goalLat, goalLon)
        # check cache for the key
        if key not in self.distances:
            self.distances[key] = dst.distance(
                (startLat, startLon), (goalLat, goalLon)).nautical
        return self.distances[key]

    def __getDirectionVector(self, previousPathElement, currentPathElement):
        curX, curY = currentPathElement
        prevX, prevY = previousPathElement

        if curX == prevX:
            if curY > prevY:
                # E
                return self.vessel.vectors["E"]
            else:
                # W
                return self.vessel.vectors["W"]
        elif curX > prevX:
            if curY == prevY:
                # N
                return self.vessel.vectors["N"]
            elif curY > prevY:
                # NE
                return self.vessel.vectors["NE"]
            else:
                # NW
                return self.vessel.vectors["NW"]
        else:
            if curY == prevY:
                # S
                return self.vessel.vectors["S"]
            elif curY > prevY:
                # SE
                return self.vessel.vectors["SE"]
            else:
                # SW
                return self.vessel.vectors["SW"]

    def enhanceGivenPathWithConsumptionData(self, path, startTimeOffset):
        mask = self.grib.file[1].values.mask
        previousPathElement = path[0]
```

```python
        travelTime = 0
        travelDistance = 0
        travelFuel = 0
        enhancedPath = [
            {
                "position": path[0],
                "time": 0,
                "distance": 0,
                "fuel": 0,
                "remainingFuel":  self.vessel.config["fuelReserve"]
            }
        ]
        # calculate the fuel consumption for every position in path regarding the vessel
        for currentPathElement in path[1:]:
            curX, curY = currentPathElement
            prevX, prevY = previousPathElement

            # if the current field is masked, the path is invalid from this point and the function ends
            if mask[curX][curY]:
                return enhancedPath

            timeIndex = self.grib.findClosestOffset(travelTime + startTimeOffset)
            direction = self.__getDirectionVector(previousPathElement, currentPathElement)

            prevLat = self.grib.lats[prevX][prevY]
            prevLon = self.grib.lons[prevX][prevY]
            curLat = self.grib.lats[curX][curY]
            curLon = self.grib.lons[curX][curY]

            distance = self.__getDistanceInNM(prevLat, prevLon, curLat, curLon)
            gribLayer = self.grib.getLayer(timeIndex)

            fuel = self.vessel.calcFuelConsumption(direction, gribLayer[curX][curY], distance)
            travelFuel = travelFuel + fuel
            travelTime = travelTime + self.__calcTime(fuel)
            travelDistance = travelDistance + distance
            enhancedPath.append({
                "position": currentPathElement,
                "time": travelTime,
                "distance": travelDistance,
                "fuel": travelFuel,
                "remainingFuel":  self.vessel.config["fuelReserve"] - travelFuel
            })
            previousPathElement = currentPathElement

        return enhancedPath

    # calculate the heuristic function by multipling the distance to the goal
        with the minimal consumption of the vessel
    def __heuristic(self, start, goal):
        startX, startY = start
        goalX, goalY = goal

        speedTW = self.vessel.config["speedTW"]
        fuelConsumption = self.vessel.config["averageFuel"]

        startLat = self.grib.lats[startX][startY]
        startLon = self.grib.lons[startX][startY]
        goalLat = self.grib.lats[goalX][goalY]
        goalLon = self.grib.lons[goalX][goalY]

        distance = self.__getDistanceInNM(startLat, startLon, goalLat, goalLon)

        return (distance/(speedTW * 2)) * fuelConsumption

    def __calcTime(self, fuel):
        return (fuel / self.vessel.config["averageFuel"]) * 60

    def __leafAStar(self, nextX, nextY, direction, openList, environmentVectors,
         mask, goal, parent):
        # expands one leaf and adds it to the open List for A* algorithm
        curX = parent["x"]
        curY = parent["y"]
        usedFuel = parent["usedFuel"]
        usedTime = parent["travelTime"]
        coveredDistance = parent["distance"]

        #is in bounds and not masked?
        if len(environmentVectors) > nextX >= 0 and len(environmentVectors[0]) >
             nextY >= 0 and mask[nextX][nextY] == False:
            curLat = self.grib.lats[curX][curY]
            curLon = self.grib.lons[curX][curY]
            nextLat = self.grib.lats[nextX][nextY]
            nextLon = self.grib.lons[nextX][nextY]

            distance = self.__getDistanceInNM(curLat, curLon, nextLat, nextLon)
            fuel = self.vessel.calcFuelConsumption(direction, environmentVectors
                [curX][curY], distance)
            time = self.__calcTime(fuel)
            currentUsedFuel = usedFuel + fuel
            if fuel != float('inf'):
                newCandidate = {
                    "x": nextX,
                    "y": nextY,
                    "usedFuel" : currentUsedFuel,
                    "validation": currentUsedFuel + self.__heuristic((nextX, nextY), goal),
                    "travelTime": usedTime + time,
                    "parent": parent,
                    "distance": coveredDistance + distance
                }
                openList.append(newCandidate)

    # calls the expand function for each cardinal point (N, NE, E, SE, S, SW, W, NW)
    def __expandAstar(self, x, y, openList, usedTime, mask, goal, parent):
        neighbors = [
            [x - 1, y, self.vessel.vectors["N"]],
            [x - 1, y + 1, self.vessel.vectors["NE"]],
            [x, y + 1, self.vessel.vectors["E"]],
            [x + 1, y + 1, self.vessel.vectors["SE"]],
            [x + 1, y, self.vessel.vectors["S"]],
            [x + 1, y - 1, self.vessel.vectors["SW"]],
            [x, y - 1, self.vessel.vectors["W"]],
            [x - 1, y - 1, self.vessel.vectors["NW"]]
        ]
        environmentVectors = self.grib.getLayer(self.grib.findClosestOffset(usedTime))
        for element in neighbors:
            x, y, direction = element
            self.__leafAStar(x, y, direction, openList, environmentVectors, mask, goal, parent)

    def __getBestFromOpenList(self, openList):
            # find nominee with the lowest needed fuel
            nomineeIndex = 0
            nomineeFuel = float('inf')
            for i in range(len(openList)):
                if openList[i]["validation"] < nomineeFuel:
                    nomineeFuel = openList[i]["validation"]
                    nomineeIndex = i
            return openList.pop(nomineeIndex)

    def astar(self, start, goal, startTimeOffset):
        # initialize the fuel Map filled with zeros
        firstGribLayer = self.grib.file[1].values
        mask = firstGribLayer.mask
        #init empty map
        fuelMap = ma.masked_array(np.zeros((len(firstGribLayer), len(firstGribLayer[0]))), dtype=float), mask=mask)
        # open and closed List as Sets to avoid duplicates
        if mask[start[0]][start[1]] or mask[goal[0]][goal[1]]:
            return {"fuelMap": fuelMap, "nominee": None}
        openList = list()
        closedList = set()
        # each tupel: (latitude, longitude, used fuel, usedFuel + heuristic, time, parent, distance)
        # add start point to openList
        first = {
            "x": start[0],
            "y": start[1],
            "usedFuel" : 0,
            "validation": self.__heuristic(start, goal),
            "travelTime": startTimeOffset,
            "parent": None,
            "distance": 0
        }
        openList.append(first)
        # loop until goal is reached or no solution possible
        while True:
            # if openList is empty, no solution was found
            if len(openList) == 0:
                return {"fuelMap": fuelMap, "nominee": None}

            #get next
            nominee = self.__getBestFromOpenList(openList)

            # expand
            if (nominee["x"], nominee["y"]) not in closedList:
                curX = nominee["x"]
                curY = nominee["y"]
                usedFuel = nominee["usedFuel"]
                usedTime = nominee["travelTime"]

                # add consumption to the fuel map
                fuelMap[curX][curY] = usedFuel

                # add node to closedList
                closedList.add((curX, curY))

                # check if goal reached
                if curX == goal[0] and curY == goal[1]:
                    return {"fuelMap": fuelMap, "nominee": nominee}

                self.__expandAstar(curX, curY, openList, usedTime, mask, goal, nominee)
```

A.1.2 grib.py

```
import pygrib
from bisect import bisect_left
```

```python
import numpy as np
import numpy.ma as ma


class Grib:
    def __init__(self, filename):
        self.filename = filename
        self.layers = dict()

        # open the GRIB file
        self.file = pygrib.open(self.filename)

        # check if Grib file is usable. For now: It contains currents (discipline = 10 and parameterCategory = 1)
        if self.file[1]['discipline'] != 10 or self.file[1]['parameterCategory'] != 1:
            raise Exception("Unsupported GRIB format")

        # get the a list of layers offsets
        self.offsets = sorted(list(set(map(lambda grib: grib['forecastTime'], self.file))))

        # The unit of time code is saved in 'stepUnits', see GRIB2 Code Table 4.4
        # https://www.nco.ncep.noaa.gov/pmb/docs/grib2/grib2_doc/grib2_table4-4.shtml (visited 07.04.22)
        layer = self.file[1]

        #found most likely a bug in pygrib
        dummy = str(layer) + str("dummy")

        unitTime = layer['stepUnits']

        if unitTime == 1:
            self.offsets = [x * 60 for x in self.offsets]
        elif unitTime == 2:
            self.offsets = [x * 1440 for x in self.offsets]
        elif unitTime == 10:
            self.offsets = [x * 180 for x in self.offsets]
        elif unitTime == 11:
            self.offsets = [x * 360 for x in self.offsets]
        elif unitTime == 12:
            self.offsets = [x * 720 for x in self.offsets]
        else:
            if unitTime != 0:
                raise Exception("Error: unknown time unit")

        self.lats, self.lons = self.file[1].latlons()

    def getLayer(self, time):
        #is it in cache?
        if time not in self.layers:
            # discipline = 10, parameterCategory = 1, parameterNumber = 2 ->
                Eastward sea water velocity
            # discipline = 10, parameterCategory = 1, parameterNumber = 3 ->
                Northward sea water velocity
            byTime = self.file.select(forecastTime=time)
            east = list(filter(lambda x: x['parameterNumber'] == 2, byTime))
            north = list(filter(lambda x: x['parameterNumber'] == 3, byTime))
            self.layers[time] = np.dstack([east[0].values, north[0].values])
        return self.layers[time]

    def findClosestOffset(self, time):
        pos = bisect_left(self.offsets, time)
        if pos == 0:
            return self.offsets[0]
        if pos == len(self.offsets):
            return self.offsets[-1]
        before = self.offsets[pos - 1]
        after = self.offsets[pos]
        if (after - time) < (time - before):
            return after
        else:
            return before

    # create a map for a calculated Path to gain a nice visualisation
    def createFuelMap(self, path, attribute):
        firstGribLayer = self.file[1].values

        # initialize the fuel Map filled with zeros
        fuelMap = ma.masked_array(np.zeros((len(firstGribLayer), len(
            firstGribLayer[0]))), dtype=float), mask=firstGribLayer.mask)
        for currentPathElement in path:
            curX, curY = currentPathElement["position"]
            fuelMap[curX][curY] = currentPathElement[attribute]
        return fuelMap
```

A.1.3 util.py

```
import numpy as np
import pickle


def angleOfEnvInfluence(directionVector, envInfluenceVector):
    angle = getAngle(directionVector, envInfluenceVector)
    return angle


```

```python
def velocityOfEnvInfluence(envInfluenceVector):
    length = np.linalg.norm(envInfluenceVector)
    return length


def courseCorrectionAngle(stw, influenceAngleDeg, influenceVelocity):
    alpha = influenceAngleDeg
    a = stw
    b = influenceVelocity
    divided = np.sin(np.deg2rad(alpha)) / a

    rad = np.arcsin(b * divided)
    deg = np.rad2deg(rad)
    return deg


def getThirdAngleOfTriangle(angle1Deg, angle2Deg):
    added = (angle1Deg + angle2Deg)
    return 180 - added


def getDTW(angleDeg, influenceAngleDeg, dog):
    """
    DTW = distance though water
    DOG = distance over ground
    """
    if angleDeg < 0.01:
        return 0

    divided = dog / np.sin(np.deg2rad(angleDeg))
    sinInfl = np.sin(np.deg2rad(influenceAngleDeg))
    return divided * sinInfl


def getTTW(dtw, stw):
    """time though water"""
    return dtw/stw

def getFuelConsumption(ttw, avgConsumption):
    return ttw * avgConsumption


def getAngle(v1, v2):
    lengthV1 = np.linalg.norm(v1)
    lengthV2 = np.linalg.norm(v2)

    isV1Small = np.isclose(lengthV1, 0)
    isV2Small = np.isclose(lengthV2, 0)
    if isV1Small or isV2Small:
        return np.nan

    unitV1 = v1 / lengthV1
    unitV2 = v2 / lengthV2
    dot_product = np.dot(unitV1, unitV2)
    rad = np.arccos(dot_product)
    deg = np.rad2deg(rad)
    return deg
```

A.1.4 vessel.py

```
import numpy as np
import pickle
import fuelcalculator as fc
from fuelcalculator import util

class Vessel:
    def __init__(self, speedTW, averageFuel, fuelReserve):
        self.config = {
            "speedTW": speedTW,
            "averageFuel": averageFuel,
            "fuelReserve": fuelReserve
        }
        # The vessel vectors result from the speed of the vessel.
        # Each vector represent the direction and speed of the vessel into 8
            cardinal points
        self.vectors = {
            "N": [0, speedTW],
            "NE": [speedTW / np.sqrt(2), speedTW / np.sqrt(2)],
            "E": [speedTW, 0],
            "SE": [speedTW / np.sqrt(2), -speedTW / np.sqrt(2)],
            "S": [0, -speedTW],
            "SW": [-speedTW / np.sqrt(2), -speedTW / np.sqrt(2)],
            "W": [-speedTW, 0],
            "NW": [-speedTW / np.sqrt(2), speedTW / np.sqrt(2)]
        }

    def save(self, fileName):
        try:
            with open(fileName, 'wb') as f:
                pickle.dump(self.config, f)
        except IOError:
            print("Error safe config")

    def getSTW(self):
        return self.config["speedTW"]

    def calcFuelConsumption(self, direction, envInfluence, distanceNM):
        #is the environmental influence stronger than the "moving force" of the
```

```
            vessel
        if np.linalg.norm(envInfluence) >= self.config["speedTW"]:
            #mark as unreachable
            return float('inf')

        influenceAngle = util.angleOfEnvInfluence(direction, envInfluence)
        if 0.1 < influenceAngle < 179.9 and not np.isnan(influenceAngle):
            stw = self.config["speedTW"]
            influenceVelocity = util.velocityOfEnvInfluence(envInfluence)
            cca = util.courseCorrectionAngle(stw, influenceAngle,
                influenceVelocity)
            angle = util.getThirdAngleOfTriangle(influenceAngle, cca)
            dog = distanceNM
            dtw = util.getDTW(angle, influenceAngle, dog)
            ttw = util.getTTW(dtw, stw)
            avgConsumption = self.config["averageFuel"]
            return util.getFuelConsumption(ttw, avgConsumption)
        else:
            cog = np.add(direction, envInfluence)
            sog = np.linalg.norm(cog)
            ttw = util.getTTW(distanceNM, sog)
            avgConsumption = self.config["averageFuel"]
            return util.getFuelConsumption(ttw, avgConsumption)

    @staticmethod
    def createVesselFromConfig(filename):
        with open(filename, 'rb') as f:
            config = pickle.load(f)
            return Vessel(config["speedTW"], config["averageFuel"], config["
                fuelReserve"])
```

A.1.5 __init__.py

```
from fuelcalculator.vessel import Vessel
from fuelcalculator.grib import Grib
from fuelcalculator.fuelcalculator import Calc
from fuelcalculator import util
```

A.2 testscripts

A.2.1 test.py

```
import unittest
import numpy as np
import fuelcalculator as fc
from fuelcalculator import util
import math
```

```python


class TestVesselMethods(unittest.TestCase):
    def testGrib(self):
        grib = fc.Grib("gribFiles/german_bay.grb2")

        self.assertEqual(grib.findClosestOffset(-1), 15, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(0), 15, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(14), 15, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(15), 15, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(16), 15, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(22), 15, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(23), 30, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(24), 30, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(254), 255, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(255), 255, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(256), 255, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(
            1439), 1440, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(
            1440), 1440, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(
            1441), 1440, 'got wrong offset')
        self.assertEqual(grib.findClosestOffset(
            54556868), 1440, 'got wrong offset')

        layer = grib.getLayer(grib.findClosestOffset(22))
        layer2 = grib.getLayer(grib.findClosestOffset(23))
        layer3 = grib.getLayer(grib.findClosestOffset(24))
        self.assertFalse(np.array_equal(layer, layer2), 'got the same layer')
        self.assertTrue(np.array_equal(layer2, layer3),
                        'got different layers but should have given the same
                            layer')

        path = [
            {
                "position": [0, 0],
                "time": 10,
                "distance": 20,
                "fuel": 30,
                "remainingFuel": 5
            },
            {
                "position": [1, 1],
                "time": 11,
                "distance": 21,
                "fuel": 31,
                "remainingFuel": 4
            },
            {
                "position": [2, 2],
                "time": 12,
                "distance": 22,
                "fuel": 32,
                "remainingFuel": 3
            },
            {
                "position": [2, 1],
                "time": 13,
                "distance": 23,
                "fuel": 33,
                "remainingFuel": 2
            },
            {
                "position": [2, 0],
                "time": 14,
                "distance": 24,
                "fuel": 34,
                "remainingFuel": 1
            },
            {
                "position": [1, 0],
                "time": 15,
                "distance": 25,
                "fuel": 35,
                "remainingFuel": 0
            }
        ]

        fuelMap = grib.createFuelMap(path, "fuel")
        timeMap = grib.createFuelMap(path, "time")
        distanceMap = grib.createFuelMap(path, "distance")
        remainingFuelMap = grib.createFuelMap(path, "remainingFuel")

        for pos in path:
            x, y = pos["position"]
            self.assertEqual(fuelMap[x, y], pos["fuel"], "fuel did not match")
            self.assertEqual(timeMap[x, y], pos["time"], "time did not match")
            self.assertEqual(
                distanceMap[x, y], pos["distance"], "distance did not match")
            self.assertEqual(
                remainingFuelMap[x, y], pos["remainingFuel"], "remainingFuel did
                     not match")

    def testVessel(self):
        speed = 5
        consumption = 2
        tank = 40
        vessel = fc.Vessel(speed, consumption, tank)

        # storage
        vessel.save("testCaseVessel.pkl")
        vesselLoaded = fc.Vessel.createVesselFromConfig("testCaseVessel.pkl")
        self.assertEqual(vessel.config, vesselLoaded.config,
                         "got different vessels")

        cases = [
            {
                "direction": "N",
                "env": [3, 8],
                "dist": 23,
                "result": float('inf')
            },
            {
                "direction": "W",
                "env": [0, 0],
                "dist": 5,
                "result": consumption
            },
            {
                "direction": "SE",
                "env": [5, 0],
                "dist": 42,
                "result": float('inf')
            },
            {
                "direction": "E",
                "env": [-4, 0],
                "dist": 5,
                "result": 10
            },
            {
                "direction": "E",
                "env": [4, 0],
                "dist": 1,
                "result": 0.22
            },
            {
                "direction": "E",
                "env": [-4, 0],
                "dist": 1,
                "result": 2
            },
            {
                "direction": "W",
                "env": [-3, 0],
                "dist": 4,
                "result": 1
            },
            {
                "direction": "N",
                "env": [3, 2],
                "dist": 10,
                "result": 3.33
            },
            {
                "direction": "N",
                "env": [-2, -1],
                "dist": 10,
                "result": 5.58
            },
            {
                "direction": "SW",
                "env": [2, -1],
                "dist": 7,
                "result": 3.66
            },
            {
                "direction": "SW",
                "env": [-3, -2],
                "dist": 7,
                "result": 1.65
            },
            {
                "direction": "NW",
                "env": [5, -1],
                "dist": 8,
                "result": float("inf")
            },
            {
                "direction": "NW",
                "env": [1, 1],
                "dist": 11,
                "result": 4.59
            },
            {
                "direction": "NW",
                "env": [2, 2],
                "dist": 11,
                "result": 5.34
            },
            {
                "direction": "SE",
                "env": [-2, -1],
                "dist": 3,
                "result": 1.57
```

```python
            },
            {
                "direction": "SE",
                "env": [-3, -5],
                "dist": 3,
                "result": float("inf")
            },
            {
                "direction": "SE",
                "env": [-3, 1],
                "dist": 6,
                "result": 6.1
            }
        ]

        for case in cases:
            case["direction"] = np.asarray(
                vessel.vectors[case["direction"]], dtype=float)
            case["env"] = np.asarray(case["env"], dtype=float)
            consumption = vessel.calcFuelConsumption(
                case["direction"], case["env"], case["dist"])
            rounded = round(consumption, 2)
            self.assertEqual(
                rounded, case["result"], "fuel consumption was expected
                    differently. input data: " + str(case))

    def testAngle(self):
        cases = [
            [[2, 3], [4, 5], 4.96974],
            [[3, 7], [5, 10], 3.36646],
            [[5, 10], [5, 10], 0],
            [[5, 10], [5, -10], 126.87],
            [[5, 10], [-5, 10], 53.1301],
            [[5, 10], [-5, -10], 180],
            [[5, 10], [-4.9, -10], 179.54],
            [[5, 10], [4.9, 10], 0.460197],
            [[5, 10], [2.5, 5], 0],
            [[5, 10], [2.5, 5], 0],
            [[1, 1], [1, -1], 90]
        ]

        for case in cases:
            self.assertEqual(round(util.getAngle(case[0], case[1]), 2), round(
                case[2], 2), "angle was not correct. input data: " + str(case))

    def testCourseCorrectionAngle(self):
        cases = [
            [0.1, 180, 0.1, 0],
            [0.2, 179, 0.2, 1],
            [0.3, 178, 0.3, 2],
            [0.4, 177, 0.4, 3],
            [0.5, 176, 0.5, 4],
            [0.6, 175, 0.6, 5],
            [0.7, 174, 0.7, 6],
            [0.8, 173, 0.8, 7],
            [0.9, 172, 0.9, 8],
            [1, 171, 1, 9],
            [1.3, 170, 1.1, 8.45],
            [1.5, 169, 1.2, 8.78],
            [1.7, 168, 1.3, 9.15],
            [2, 167, 1.4, 9.06],
            [2.2, 166, 1.5, 9.49],
            [2.8, 165, 1.6, 8.51],
            [3, 164, 1.7, 8.99],
            [3.33, 163, 1.8, 9.09],
            [3.7, 162, 1.9, 9.13],
            [3.9, 161, 2, 9.61],
            [4.01, 160, 2.1, 10.32],
            [4.6, 159, 2.2, 9.87],
            [5, 158, 2.3, 9.92],
            [7, 157, 2.4, 7.7],
            [10, 156, 2.5, 5.84],
            [0.1, 0.1, 2.6, 2.6],
            [0.2, 0.2, 2.7, 2.7],
            [0.3, 0.3, 2.8, 2.8],
            [0.4, 0.4, 2.9, 2.9],
            [0.5, 0.5, 3, 3],
            [0.6, 0.6, 3.1, 3.1],
            [0.7, 0.7, 3.2, 3.2],
            [0.8, 0.8, 3.3, 3.3],
            [0.9, 0.9, 3.4, 3.4],
            [1, 1, 3.5, 3.5],
            [1.3, 1.1, 3.6, 3.05],
            [1.5, 1.2, 3.7, 2.96],
            [1.7, 1.3, 3.8, 2.91],
            [2, 1.4, 3.9, 2.73],
            [2.2, 1.5, 4, 2.73],
            [2.8, 1.6, 4.1, 2.34],
            [3, 1.7, 4.2, 2.38],
            [3.33, 1.8, 4.3, 2.32],
            [3.7, 1.9, 4.4, 2.26],
            [3.9, 2, 4.5, 2.31],
            [4.01, 2.1, 4.6, 2.41],
            [4.6, 2.2, 4.7, 2.25],
            [5, 2.3, 4.8, 2.21],
            [7, 2.4, 4.9, 1.68],
            [10, 93.2, 7.5, 48.49],
            [0.1, 179.2, 0, 0],
            [0.2, 179.3, 0.1, 0.35],
            [0.3, 179.4, 0.2, 0.4],
            [0.4, 179.5, 0.3, 0.37],
            [0.5, 179.6, 0.4, 0.32],
            [0.6, 179.7, 0.5, 0.25],
            [0.7, 179.8, 0.6, 0.17],
            [0.8, 179.9, 0.7, 0.09],
            [0.9, 180, 0.8, 0],
            [1, 179.9, 0.9, 0.09],
            [1.3, 179.8, 1, 0.15],
            [1.5, 179.7, 1.1, 0.22],
            [1.7, 179.6, 1.2, 0.28],
            [2, 179.5, 1.3, 0.32],
            [2.2, 179.4, 1.4, 0.38],
            [2.8, 179.3, 1.5, 0.37],
            [3, 179.2, 1.6, 0.43],
            [3.33, 179.1, 1.7, 0.46],
            [3.7, 179, 1.8, 0.49],
            [3.9, 178.9, 1.9, 0.54],
            [4.01, 178.8, 2, 0.6],
            [4.6, 178.7, 2.1, 0.59],
            [5, 178.6, 2.2, 0.62],
            [7, 178.5, 2.3, 0.49],
            [10, 178.4, 2.4, 0.38],
            [0.1, 180, 0.01, 0],
            [0.2, 180, 0.02, 0],
            [0.3, 180, 0.03, 0],
            [0.4, 180, 0.04, 0],
            [0.5, 180, 0.05, 0],
            [0.6, 180, 0.06, 0],
            [0.7, 180, 0.07, 0],
            [0.8, 180, 0.08, 0],
            [0.9, 180, 0.09, 0],
            [1, 180, 0.1, 0],
            [1.3, 180, 0.11, 0],
            [1.5, 180, 0.1, 0],
            [1.7, 180, 0.2, 0],
            [2, 180, 0.3, 0],
            [2.2, 180, 0.4, 0],
            [2.8, 180, 0.5, 0],
            [3, 180, 0.6, 0],
            [3.33, 180, 0.7, 0],
            [3.7, 180, 0.8, 0],
            [3.9, 180, 0.9, 0],
            [4.01, 180, 1, 0],
            [4.6, 180, 1.1, 0],
            [5, 180, 1.2, 0],
            [7, 180, 1.3, 0],
            [10, 180, 1.4, 0],
            [3.33, 166, 3.2, 13.44],
            [3.7, 167, 3.3, 11.57],
            [3.9, 168, 3.4, 10.44],
            [4.01, 169, 3.5, 9.59],
            [4.6, 170, 3.6, 7.81],
            [5, 171, 3.7, 6.65],
            [7, 172, 3.8, 4.33],
            [10, 173, 3.9, 2.72],
            [0.5, 174, 4.4, 66.9],
            [0.6, 175, 4.5, 40.82],
            [0.7, 176, 4.6, 27.28],
            [0.8, 177, 4.7, 17.91],
            [0.9, 178, 4.8, 10.73],
            [1, 179, 4.9, 4.91],
            [1.3, 180, 5, 0],
            [1.5, 0, 5.1, 0],
            [1.7, 1, 5.2, 3.06],
            [2, 2, 5.3, 5.31],
            [2.2, 3, 5.4, 7.38],
            [2.8, 4, 5.5, 7.88],
            [3, 5, 5.6, 9.36],
            [3.33, 6, 5.7, 10.31],
            [3.7, 7, 5.8, 11.01],
            [3.9, 8, 5.9, 12.15],
            [4.01, 9, 6, 13.54],
            [4.6, 10, 6.1, 13.31],
            [5, 11, 6.2, 13.69],
            [7, 12, 6.3, 10.78],
            [10, 13, 6.4, 8.28]
        ]

        for case in cases:
            stw, influenceAngleDeg, influenceVelocity, expected = case
            res = util.courseCorrectionAngle(
                stw, influenceAngleDeg, influenceVelocity)
            res = round(res, 2)
            expected = round(expected, 2)

            self.assertEqual(
                res, expected, "angle was not correct. input data: " + str(case)
                    )

    def testDTW(self):
        cases = [
            [0, 150, 0.2, 0],
            [0.01, 147, 0.4, 1248.22],
            [0.2, 144, 0.6, 101.03],
            [2, 141, 0.8, 14.43],
            [4, 138, 1, 9.59],
            [6, 135, 1.2, 8.12],
            [8, 132, 1.4, 7.48],
            [10, 129, 1.6, 7.16],
            [12, 126, 1.8, 7],
            [14, 123, 2, 6.93],
            [16, 120, 2.2, 6.91],
            [18, 117, 2.4, 6.92],
            [20, 114, 2.6, 6.94],
            [22, 111, 2.8, 6.98],
            [24, 108, 3, 7.01],
            [26, 105, 3.2, 7.05],
            [28, 102, 3.4, 7.08],
            [30, 99, 3.6, 7.11],
            [32, 96, 3.8, 7.13],
            [34, 93, 4, 7.14],
            [36, 90, 4.2, 7.15],
            [38, 87, 4.4, 7.14],
            [40, 84, 4.6, 7.12],
            [42, 81, 4.8, 7.09],
            [44, 78, 5, 7.04],
            [46, 75, 50, 67.14],
            [48, 72, 40, 51.19],
            [50, 69, 30, 36.56],
            [52, 66, 20, 23.19],
            [54, 63, 10, 11.01],
            [56, 60, 0, 0],
            [58, 57, 1, 0.99],
            [60, 54, 2, 1.87],
            [62, 51, 3, 2.64],
            [64, 48, 4, 3.31],
            [66, 45, 5, 3.87],
            [68, 42, 6, 4.33],
            [70, 39, 7, 4.69],
            [72, 36, 8, 4.94],
            [74, 33, 9, 5.1],
            [76, 30, 10, 5.15],
            [78, 27, 11, 5.11],
            [80, 24, 12, 4.96],
            [82, 21, 13, 4.7],
            [84, 18, 14, 4.35],
            [86, 15, 15, 3.89],
            [88, 12, 16, 3.33],
            [90, 9, 17, 2.66],
            [92, 6, 18, 1.88],
            [94, 3, 19, 1],
            [96, 0, 20, 0],
            [98, 0.01, 21, 0],
            [100, 0.2, 22, 0.08],
            [102, 75, 23, 22.71],
            [104, 76, 24, 24],
            [106, 10, 25, 4.52],
            [108, 11, 26, 5.22],
            [110, 12, 27, 5.97],
            [112, 13, 28, 6.79],
            [114, 14, 29, 7.68],
            [116, 15, 30, 8.64],
            [118, 16, 31, 9.68],
            [120, 17, 32, 10.8],
            [122, 18, 33, 12.02],
            [124, 19, 34, 13.35],
            [126, 20, 35, 14.8],
            [128, 21, 36, 16.37],
            [130, 22, 37, 18.09],
            [132, 23, 38, 19.98],
            [134, 24, 39, 22.05],
            [136, 25, 40, 24.34],
            [138, 26, 41, 26.86],
            [140, 27, 42, 29.66],
            [142, 28, 43, 32.79],
            [144, 29, 44, 36.29],
            [146, 30, 45, 40.24],
            [148, 31, 46, 44.71],
            [150, 15.2, 47, 24.65],
            [152, 14.3, 48, 25.25],
            [154, 13.4, 49, 25.9],
            [156, 12.5, 50, 26.61],
            [158, 11.6, 51, 27.38],
            [160, 10.7, 52, 28.23],
            [162, 9.8, 53, 29.19],
            [164, 8.9, 54, 30.31],
            [166, 8, 55, 31.64],
            [168, 7.1, 56, 33.29],
            [170, 6.2, 57, 35.45],
            [172, 5.3, 58, 38.5],
            [174, 4.4, 59, 43.3],
            [176, 3.5, 60, 52.51],
            [178, 0.5, 61, 15.25],
            [179, 0.4, 62, 24.8],
            [180, 0, 63, 0]
        ]

        for case in cases:
            angleDeg, influenceAngleDeg, dog, expected = case

            res = util.getDTW(angleDeg, influenceAngleDeg, dog)
            res = round(res, 2)
            expected = round(expected, 2)

            self.assertEqual(
                res, expected, "distance was not correct. input data: " + str(
                    case))

if __name__ == '__main__':
    unittest.main()
```

A.2.2 testFuelConsumption.py

```
import fuelcalculator as fc
from testArrays import *


def runTest(vessel, testArray, fuelArray):
    for x in range(len(testArray)):
        for y in range(len(testArray[0])):
            fuelArray[x][y] = vessel.calcFuelConsumption(
                vessel.vectors["E"], testArray[x][y], 0.5)
    return fuelArray


def saveImage(inputs, results, color='Blues'):
    viridis = cm.get_cmap('viridis', 256)
    newcolors = viridis(np.linspace(0, 1, 256))
    white = np.array([256, 256, 256, 1])
    newcolors[:25, :] = white
    newcmp = ListedColormap(newcolors)

    fig = plt.figure(figsize=(20, 20))

    pos = 1
    for data, result in zip(inputs, results):
        plot(pos, result, data, 'Test ' + str(pos), color)
        pos += 1

    plt.show
    plt.savefig('output/testFuelConsumption.png')


def plot(position, fuelData, mapData, title, color):
    plt.subplot(2, 2, position)
    plt.pcolor(fuelData, vmin=0.1, vmax=0.5, cmap=plt.cm.get_cmap(color))
    bar = plt.colorbar()
    plt.title(title, fontdict={'fontsize': 30})
    plt.gca().set_aspect('equal')
    printSmallArray(mapData)
    bar.set_label('Fuel consumption (l per field)', fontdict={'fontsize': 20})


```

```
def runTests():

    #cmaps = ["Blues", "afmhot", "autumn", "binary", "OrRd", "PuBu", "PuBuGn", "
        Purples", "rainbow", "RdYlBu", "Reds", "turbo", "YlGnBu", "YlOrRd"]

    # create a example vessel, which drives with 5kn, consumes 2l per hour and
        has a 80l tank
    sampleVessel = fc.Vessel(5, 2, 80)

    # create test arrays, which represent the environmental influences
    inputs = []
    inputs.append(createTestArray1Small())
    inputs.append(createTestArray2Small())
    inputs.append(createTestArray3Small())
    inputs.append(createTestArray4Small())

    # run the tests
    results = []
    for test in inputs:
        results.append(runTest(sampleVessel, test,
                               np.zeros((11, 11), dtype=float)))

    #for colormap in cmaps:
        # save the tests as image
    #saveImage(inputs, results, colormap)
    saveImage(inputs, results, "YlGnBu")

print("test 'calcFuel' started, this may take some while")
runTests()
```

A.2.3 testPredeterminedPath.py

```
from unittest import result
import fuelcalculator as fc
from testArrays import *
import time

# function to measure the time
def countTime(path, offset, testObject, liste):
    start = time.time()
    liste.append(testObject.enhanceGivenPathWithConsumptionData(path, offset))
    end = time.time()
    return end - start

# clear the cashe for each run and count the time needed to calculate the path
def runTest(path, startOffset, calculators, name):
    times = list()
    results = list()
    for element in calculators:
```

```python
        element.distances = dict()
        element.grib.layers = dict()
        times.append(countTime(path, startOffset, element, results))
    print("--------Test " + name + ":----------")
    print("min:", min(times))
    print("max:", max(times))
    print("average:", sum(times) / len(times))
    firstEnhancedPath = results[0]
    lastPathElement = firstEnhancedPath[-1]
    print("travelTime:",lastPathElement["time"])
    print("distance:", lastPathElement["distance"])
    print("fuel:", lastPathElement["fuel"])
    grib = fc.Grib('gribFiles/german_bay.grb2')
    fuelMap = grib.createFuelMap(firstEnhancedPath, "fuel")
    saveTest(name, fuelMap)

def saveTest(titel, data):
    plt.figure(figsize = (30,20))
    plt.pcolor(data)
    bar = plt.colorbar()
    plt.title(titel)
    plt.gca().invert_yaxis()
    plt.gca().set_aspect('equal')
    bar.set_label('consumed fuel (l per field)', fontdict= {'fontsize' : 20})
    plt.savefig('output/preDeterminedPath_' + titel + '.png')

def createTestCalculatorObjects(count):
    test = list()
    for x in range(count):
        test.append(fc.Calc(fc.Vessel(5, 2, 40), fc.Grib('gribFiles/german_bay.grb2')))
    return test

# definde two test paths, from Borkum to Norderney and from Borkum to Cuxhaven
borkumNorderney = [[345, 39], [346, 39], [346, 38], [346, 37], [345, 36],
                    [344, 35], [343, 35], [342, 35], [341, 35], [340, 35],
                    [339, 36], [338, 37], [337, 38], [336, 39], [335, 40],
                    [335, 41], [334, 42], [334, 43], [334, 44], [334, 45],
                    [333, 46], [333, 47], [332, 48], [332, 49], [332, 50],
                    [331, 51], [331, 52], [331, 53], [330, 54], [330, 55],
                    [330, 56], [330, 57], [329, 58], [329, 59], [329, 60],
                    [329, 61], [329, 62], [329, 63], [329, 64], [329, 65],
                    [329, 66], [329, 67], [330, 68], [331, 69], [331, 70],
                    [331, 71], [330, 72], [329, 72]]

borkumCuxhaven = [[345, 39], [346, 39], [346, 38], [346, 37], [345, 36],
                   [344, 35], [343, 35], [342, 35], [341, 35], [340, 35],
                   [339, 36], [338, 37], [337, 38], [336, 39], [335, 40],
                   [334, 41], [334, 42], [333, 43], [333, 44], [333, 45],
                    [332, 46], [332, 46], [332, 47], [331, 48], [331, 49],
                    [331, 50], [330, 51], [330, 52], [330, 53], [329, 54],
                    [329, 55], [329, 56], [329, 57], [328, 58], [328, 59],
                    [328, 60], [328, 61], [327, 62], [327, 63], [327, 64],
                    [327, 65], [326, 66], [326, 67], [326, 68], [326, 69],
                    [325, 70], [325, 71], [325, 72], [325, 73], [324, 74],
                    [324, 75], [324, 76], [324, 77], [323, 78], [323, 79],
                    [323, 80], [322, 81], [322, 82], [322, 83], [321, 84],
                    [321, 85], [321, 86], [320, 87], [320, 88], [320, 89],
                    [319, 90], [319, 91], [319, 92], [318, 93], [318, 94],
                    [318, 95], [317, 96], [317, 97], [317, 98], [316, 99],
                    [316, 100], [316, 101], [315, 102], [315, 103], [315, 104],
                    [314, 105], [314, 106], [314, 107], [313, 108], [313, 109],
                    [313, 110], [312, 111], [312, 112], [312, 113], [311, 114],
                    [311, 115], [311, 116], [310, 117], [310, 118], [310, 119],
                    [309, 120], [309, 121], [309, 122], [309, 123], [308, 124],
                    [308, 125], [308, 126], [308, 127], [307, 128], [307, 129],
                    [307, 130], [307, 131], [306, 132], [306, 133], [306, 134],
                    [306, 135], [305, 136], [305, 137], [305, 138], [305, 139],
                    [304, 140], [304, 141], [304, 142], [304, 143], [303, 144],
                    [303, 145], [303, 145], [303, 146], [302, 147], [302, 148],
                    [302, 149], [302, 150], [301, 151], [301, 152], [301, 153],
                    [301, 154], [300, 155], [300, 156], [300, 157], [300, 158],
                    [299, 159], [299, 160], [299, 161], [299, 162], [298, 163],
                    [298, 164], [298, 165], [298, 166], [298, 167], [298, 168],
                    [298, 169], [298, 170], [298, 171], [298, 172], [299, 173],
                    [299, 174], [300, 175], [301, 176], [302, 177], [303, 178],
                    [304, 179], [305, 180], [306, 181], [307, 182], [308, 182]]

print("test 'predeterminedPath' started, this may take some while")

runTest(borkumNorderney, 340, createTestCalculatorObjects(20), 'borkum-
    Norderney_good_conditions')
runTest(borkumNorderney, 0, createTestCalculatorObjects(20), 'borkum-
    Norderney_bad_conditions')
runTest(borkumCuxhaven, 340, createTestCalculatorObjects(20), 'borkum-
    Cuxhaven_good_conditions')
runTest(borkumCuxhaven, 0, createTestCalculatorObjects(20), 'borkum-
    Cuxhaven_bad_conditions')
```

A.2.4 testAstar.py

```
import fuelcalculator as fc
from testArrays import *
import time

def saveTest(resultAstar, titel):
    printMap(printAstarPath(resultAstar["fuelMap"], resultAstar["nominee"]),
        titel)
```

```python

# function to measure the time
def countTime(start, goal, offset, testObject):
    startTime = time.time()
    result = testObject.astar(start, goal, offset)
    endTime = time.time()
    return endTime - startTime, result

def printResults(candidate, runtime, time):
    print("--------------------------------------------")
    print(str(time) + " min, runtime:", runtime)
    print(str(time) + " min, fuel:", candidate["nominee"]["usedFuel"])
    print(str(time) + " min, time:", candidate["nominee"]["travelTime"] - time)


def runTest():
    # test points
    borkum = (346,37)
    norderney = (331,75)
    cuxhaven = (310,185)
    sylt = (180,170)

    calculator = fc.Calc(fc.Vessel(5, 2, 80), fc.Grib('gribFiles/german_bay.grb2
        '))

    # test the same route for 12h each hour
    offset = 60
    aStarResults = list()
    for x in range(12):
        time, result = countTime(borkum, norderney, offset, calculator)
        aStarResults.append([result, time])
        offset += 60
    time = 60
    for aStarResult, runtime in aStarResults:
        printResults(aStarResult, runtime, time)
        time +=60

    # print a few A stars

    saveTest(calculator.astar(borkum, norderney, 340), 'output/
        borkum_to_norderney_start340')
    saveTest(calculator.astar(borkum, norderney, 800), 'output/
        borkum_to_norderney_start800')
    saveTest(calculator.astar(borkum, cuxhaven, 0), 'output/
        borkum_to_cuxhaven_start0')
    saveTest(calculator.astar(borkum, cuxhaven, 340), 'output/
        borkum_to_cuxhaven_start340')
    saveTest(calculator.astar(cuxhaven, sylt, 0), 'output/
        cuxhaven_to_sylt_start0')
```

```python

print("test 'A*' started, this may take some while")
runTest()
```

A.2.5 testArrays.py

```python
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.pyplot import figure
from matplotlib import cm
from matplotlib.colors import ListedColormap, LinearSegmentedColormap


def createTestArray1Small():
    # Array1
    array1line = np.linspace(-3, 3, 11, dtype=float)
    array1line2d = array1line.reshape((1, 11))
    array1East = np.repeat(array1line2d, repeats=11, axis=0)
    array1North = np.zeros((11, 11), dtype=float)
    testArray1 = np.dstack([array1East, array1North])
    return testArray1


def createTestArray2Small():
    # Array2
    array2line = np.linspace(3, -3, 11, dtype=float)
    array2North1d = np.repeat(array2line, repeats=11)
    array2North = array2North1d.reshape((11, 11))
    array2East = np.zeros((11, 11), dtype=float)
    testArray2 = np.dstack([array2East, array2North])
    return testArray2


def createTestArray3Small():
    # Array3
    testArray3 = np.zeros((11, 11, 2), dtype=float)
    for x in range(11):
        for y in range(11):
            testArray3[y][x][0] = (5-x)*(2/5)
            testArray3[y][x][1] = (5-y)*(2/5)
    return testArray3


def createTestArray4Small():
    testArray4 = np.zeros((11, 11, 2), dtype=float)
    for x in range(11):
        for y in range(11):
            testArray4[y][x][0] = -(5-x)*(2/5)
            testArray4[y][x][1] = -(5-y)*(2/5)
    return testArray4
    # Array4


def printSmallArray(testArray):
    plt.quiver(testArray.reshape((−1, 2)).T[0].reshape((11, 11)), testArray.
        reshape(
        (−1, 2)).T[1].reshape((11, 11)), units='xy', pivot='tail', scale=2,
            headwidth=7)
    plt.xlim(−2, 12)
    plt.ylim(−2, 12)

def printMap(data, titel):
    plt.figure(figsize = (30,20))
    plt.pcolor(data)
    bar = plt.colorbar()
    plt.title(titel, fontdict= {'fontsize' : 30})
    plt.gca().invert_yaxis()
    plt.gca().set_aspect('equal')
    bar.set_label('consumed fuel (l per field)', fontdict= {'fontsize' : 20})
    plt.savefig(titel + '.png')

def printAstarPath(fuelMap, winner):
    fuelMap[winner["x"]][winner["y"]] = 0
    # traverse path in reserved order to draw the path to the colored fuel map
    next = winner["parent"]
    while next != None:
        fuelMap[next["x"]][next["y"]] = 0
        next = next["parent"]
    return fuelMap
```

A.3 GUI

A.3.1 server.py

```
from __future__ import print_function
from testArrays import *
import math
import fuelcalculator as fc
import numpy as np
import time
from flask import Flask, request
from flask import jsonify
from flask import Flask, render_template
from flask import send_from_directory
import glob

import gpxpy
```

```python
import gpxpy.gpx
import base64
import datetime

from fuelcalculator import grib
app = Flask(__name__)


app = Flask(__name__)

gribCache = dict()
calcCache = dict()


@app.route("/")
def sendIndex():
    return send_from_directory('static', 'index.html')


@app.route('/<path:path>')
def sendStatic(path):
    return send_from_directory('static', path)


@app.route('/api/calc/astar', methods=['POST'])
def calcAstar():
    request_data = request.get_json()

    speed = request_data["speed"]
    consumption = request_data["consumption"]
    fileName = request_data["fileName"]
    astar = request_data["astar"]
    tank = request_data["tank"]
    time = request_data["time"]

    vessel = fc.Vessel(speed, consumption, tank)
    calc = getCalculatorFromCache(vessel, fileName)

    start = [astar["start"]["y"], astar["start"]["x"]]
    goal = [astar["goal"]["y"], astar["goal"]["x"]]

    result = calc.astar(start, goal, time)
    data = dict()
    if result["nominee"] is None:
        fuelMap = result["fuelMap"]
        data["log"] = "goal was unreachable"
    else:
        fuelMap = printAstarPath(result["fuelMap"], result["nominee"])

        data["gpx"] = createGPX(getLatLonPathFromFromAstart(result["nominee"],
            calc.grib))

        travelTime = round(result["nominee"]["travelTime"] - time, 2)
        travelDistance = round(result["nominee"]["distance"], 2)
        fuel = round(result["nominee"]["usedFuel"], 2)
        remFuel = round(tank - fuel, 2)

        data["log"] = "fuel: " + str(fuel) + " liters, remaining fuel: " + str(
            remFuel) + " liters, travel time: " + str(travelTime) + " minutes,
                distance: " + str(travelDistance) + " NM"

    width, height, pixels = getDataFromGribMap(fuelMap)
    data["pixels"] = pixels
    data["height"] = height
    data["width"] = width

    return jsonify(data)


@app.route('/api/calc/path', methods=['POST'])
def calcPath():
    request_data = request.get_json()

    speed = request_data["speed"]
    consumption = request_data["consumption"]
    fileName = request_data["fileName"]
    path = request_data["path"]
    tank = request_data["tank"]
    time = request_data["time"]

    vessel = fc.Vessel(speed, consumption, tank)
    calc = getCalculatorFromCache(vessel, fileName)

    path = [[int(element["y"]), int(element["x"])] for element in path]
    result = calc.enhanceGivenPathWithConsumptionData(path, time)

    travelTime = round(result[-1]["time"], 2)
    travelDistance = round(result[-1]["distance"], 2)
    fuel = round(result[-1]["fuel"], 2)
    remFuel = round(tank - fuel, 2)

    grib = getGribFromCache(fileName)["object"]
    fuelMap = grib.createFuelMap(result, "fuel")

    width, height, pixels = getDataFromGribMap(fuelMap)

    data = {"log": "fuel: " + str(fuel) + " liters, remaining fuel: " + str(
        remFuel) + " liters, travel time: " +
```

```python
            str(travelTime) + " minutes, distance: " + str(travelDistance) + " NM"}
    data["pixels"] = pixels
    data["height"] = height
    data["width"] = width
    data["gpx"] = createGPX(getLatLonPathFromPathObject(result, calc.grib))

    return jsonify(data)


@app.route("/api/grib/list")
def getListOfGribFileNames():
    path = r'gribFiles/*.grb2'
    return jsonify(glob.glob(path))


@app.route("/api/grib/load/<path:path>")
def getGribData(path):
    grib = getGribFromCache(path)
    return jsonify(grib["data"])


def getCalculatorFromCache(vessel, gribFileName):
    if gribFileName not in calcCache:
        grib = getGribFromCache(gribFileName)
        calc = fc.Calc(vessel, grib["object"])
        calcCache[gribFileName] = calc
    return calcCache[gribFileName]


def getGribFromCache(path):
    if path not in gribCache:
        gribCache[path] = dict()
        grib = fc.Grib(path)
        gribCache[path]["object"] = grib
        data = dict()

        someFile = grib.file[1]

        data["year"] = someFile["year"]
        data["month"] = someFile["month"]
        data["day"] = someFile["day"]
        data["hour"] = someFile["hour"]
        data["minute"] = someFile["minute"]
        data["second"] = someFile["second"]

        layer = someFile.values
        width, height, pixels = getDataFromGribMap(layer, maskOnly=True)

        data["height"] = height
        data["width"] = width
        data["pixels"] = pixels
        data["times"] = grib.offsets

        gribCache[path]["data"] = data

    return gribCache[path]


def getDataFromGribMap(layer, maskOnly=False):
    height, width = np.shape(layer)
    mask = layer.mask

    layermax = np.amax(layer)

    pixels = list()
    for x in range(height):
        row = list()
        for y in range(width):
            try:
                if mask[x][y]:
                    row.append([0, 0, 0])
                else:
                    if maskOnly:
                        row.append([255, 255, 255])
                    else:
                        red, green, blue = floatRgb(layer[x][y], 0, layermax)
                        row.append(
                            [int(red * 255), int(green * 255), int(blue * 255)])
            except Exception as err:
                pass
        pixels.append(row)

    return width, height, pixels


def floatRgb(mag, cmin, cmax):
    """ Return a tuple of floats between 0 and 1 for R, G, and B. """
    # Normalize to 0-1
    try:
        x = float(mag-cmin)/(cmax-cmin)
    except ZeroDivisionError:
        x = 0.5  # cmax == cmin
    if math.isnan(x):
        x = 0.5
    blue = min((max((4*(0.75-x), 0.)), 1.))
    red = min((max((4*(x-0.25), 0.)), 1.))
    green = min((max((4*math.fabs(x-0.5)-1., 0.)), 1.))
    return red, green, blue

def getLatLonPathFromFromAstart(goal, grib):
    path = list()

    curPoint = goal
    while curPoint:
        path.append({
            "position": [curPoint["x"], curPoint["y"]],
            "time": curPoint["travelTime"],
            "distance": curPoint["distance"],
            "fuel": curPoint["usedFuel"],
            "remainingFuel":  0
        })
        curPoint = curPoint["parent"]

    path.reverse()
    return getLatLonPathFromPathObject(path, grib)


def getLatLonPathFromPathObject(path, grib):
    for point in path:
        x = point["position"][0]
        y = point["position"][1]
        point["lat"] = grib.lats[x][y]
        point["lon"] = grib.lons[x][y]
    return path


def createGPX(path):
    gpx = gpxpy.gpx.GPX()

    isodatestr = str(datetime.datetime.utcnow().isoformat())

    gpx_route = gpxpy.gpx.GPXRoute(name="Import Fuel Calculator " + isodatestr)
    gpx.routes.append(gpx_route)

    # Create points:
    for point in path:
        gpx_route.points.append(gpxpy.gpx.GPXRoutePoint(point["lat"], point["lon
            "], elevation=0, description="estimated fuel: " + str(point["fuel"])
             + " liters"))

    xmlString = gpx.to_xml()
    base64Bytes = base64.b64encode(xmlString.encode("utf8"))
    base64String = base64Bytes.decode("ascii")

    return 'data:application/gpx+xml;base64,' + base64String

```

```

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```

A.3.2 index.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="css/bootstrap.min.css" rel="stylesheet">

    <title>FuelCalculator</title>

    <style>
        #overlay {
            position: fixed;
            display: none;
            width: 100%;
            height: 100%;
            top: 0;
            left: 0;
            right: 0;
            bottom: 0;
            background-color: rgba(0, 0, 0, 0.5);
            z-index: 2;
        }

        #spinner {
            position: absolute;
            top: 50%;
            left: 50%;
            text-align: center;
            transform: translateX(-50%);
        }
    </style>
</head>

<body>
    <div class="container-fluid">
        <div id="overlay">
            <div id="spinner">
                <div class="spinner-border" role="status" style="width: 10rem;
                    height: 10rem;">
                </div>
            </div>
        </div>
        <div class="row mb-3">
            <div class="col">
                <canvas class="img" id="gribViewer" width="200" height="100"></
                    canvas>
            </div>
            <div class="col">
                <div class="row mb-3">
                    <h1><br>FuelCalculator</h1>
                </div>
                <div class="row mb-3">
                    <div class="input-group">
                        <div class="input-group-prepend">
                            <label class="input-group-text" for="gribFileSelect"
                                >file &#x1F4C2;</label>
                        </div>
                        <select id="gribFileSelect" class="form-select custom-
                            select" disabled>
                            <option selected value="none"></option>
                        </select>
                    </div>
                </div>
                <div class="row mb-3">
                    <div class="col">
                        <div class="input-group">
                            <div class="input-group-prepend">
                                <label class="input-group-text" for="
                                    gribTimeSelect">time &#x23F0;</label>
                            </div>
                            <select id="gribTimeSelect" class="form-select
                                custom-select" disabled>
                            </select>
                        </div>
                    </div>
                    <div class="col">
                        <div class="input-group flex-nowrap">
                            <span class="input-group-text">zoom &#x1F50E;</span>
                            <input type="number" id="zoom" class="form-control"
                                min="1" max="10" value="2" step="0.5"
                                disabled>
                        </div>
                    </div>
                </div>
                <div class="row mb-3">
                    <div class="col ">
                        <button type="button" id="buttonPath" class="btn btn-
                            primary btn-lg" disabled>path &#x1F4C8;</button>
                    </div>
                    <div class="col">
                        <button type="button" id="buttonAstar" class="btn btn-
```

```
                            primary btn-lg" disabled>A* &#x2B50;</button>
                    </div>
                    <div class="col">
                        <button type="button" id="buttonCalc" class="btn btn-primary btn-lg" disabled>calculate &#x1F5A9;</button>
                    </div>
                    <div class="col">
                        <a class="btn btn-primary btn-lg disabled" id="download" href="#" role="button" download="fuelCalculator.gpx" >download &#x2B07;&#xFE0F;</a>
                    </div>
                </div>
                <div class="row mb-3">
                    <div class="col">
                        <div class="input-group flex-nowrap">
                            <span class="input-group-text">speed &#x1F4A8;</span>
                            <input type="number" id="vesselSpeed" class="form-control" min="1" max="10" value="5"
                                step="1">
                        </div>
                    </div>
                    <div class="col">
                        <div class="input-group flex-nowrap">
                            <span class="input-group-text">consumption &#x26FD;</span>
                            <input type="number" id="vesselConsumption" class="form-control" min="1" max="100" value="2"
                                step="0.25">
                        </div>
                    </div>
                    <div class="col">
                        <div class="input-group flex-nowrap">
                            <span class="input-group-text">tank &#x1F50B;</span>
                            <input type="number" id="vesselTank" class="form-control" min="5" max="100" value="40"
                                step="5">
                        </div>
                    </div>
                </div>
                <div class="row mb-3">
                    <div class="input-group form-floating">
                        <textarea class="form-control" id="outputField" rows="20"></textarea>
                        <label for="outputField" class="form-label">output &#x1F4DD;</label>
                    </div>
                </div>
```

```
            </div>
        </div>
        <div class="row mb-3">&#x00A9;&#xFE0F; made by Antonia Mueller-Baumgart</div>
    </div>
    <script src="js/jquery-3.6.0.min.js"></script>
    <script src="js/bootstrap.bundle.min.js"></script>
    <script src="js/main.js"></script>
</body>

</html>
```

A.3.3 main.js

```
let currentGrib;
let astarPositions;
let path;
let astarOrPath;
let pathColor = "green"
let pointColor = "red"

$(document).ready(function () {
    reset();
    getGribs();

    $("#gribFileSelect").change(function () {
        loadGrib(this.value);
    });

    $("#buttonPath").click(function () {
        resetPathAndAstar();
        astarOrPath = "path";
        $("#gribViewer").click(function (event) {
            let pos = getMousePos(this, event);
            drawPointOnGribView(pos, 2, pointColor);
            if (!Array.isArray(path) || path.length == 0) {
                path = [pos];
            }
            else {
                addElementsToPath(pos);
                $("#buttonCalc").prop("disabled", false);
            }
        });
    });

    $("#buttonAstar").click(function () {
        resetPathAndAstar();
        astarOrPath = "astar";
        $("#gribViewer").click(function (event) {
            let pos = getMousePos(this, event);
            drawPointOnGribView(pos, 2, pointColor);
            if (!astarPositions["start"]) {
                astarPositions["start"] = pos;
            }
            else {
                astarPositions["goal"] = pos;
                $("#gribViewer").off();
                $("#buttonCalc").prop("disabled", false);
            }
        });
    });

    $("#buttonCalc").click(function () {
        $("#overlay").show();
        data = {
            "speed": parseInt($("#vesselSpeed").val()),
            "consumption": parseInt($("#vesselConsumption").val()),
            "tank": parseInt($("#vesselTank").val()),
            "time": parseInt($("#gribTimeSelect").val()),
            "fileName": $("#gribFileSelect").val(),
            "astar": astarPositions,
            "path": path
        };
        $.ajax({
            url: "/api/calc/" + astarOrPath,
            type: 'POST',
            data: JSON.stringify(data),
            contentType: 'application/json',
            dataType: 'json',
            success: function (fuelMap) {
                drawGribView(fuelMap);
                old = $("#outputField").val();
                $("#outputField").val("[" + astarOrPath + "] " + fuelMap["log"]
                    + "\n" + old);
                if (fuelMap["gpx"]) {
                    $("#download").prop("href", fuelMap["gpx"]);
                    $("#download").removeClass("disabled");
                }
            }
        })
            .fail(function () {
                old = $("#outputField").val();
                $("#outputField").val("[" + astarOrPath + "] no answer from
                    Server\n" + old);
            })
            .always(function () {
                $("#overlay").hide();
            });
    });

    $("#zoom").change(function () {
        setGribViewSize(currentGrib["width"], currentGrib["height"]);
    });

});

function resetPathAndAstar() {
    astarOrPath = "";
    $("#buttonCalc").prop("disabled", true);
    $("#download").addClass("disabled");
    $("#download").prop("href", "#");
    drawGribView(currentGrib);
    astarPositions = {
        "start": null,
        "goal": null
    };
    path = [];
    $("#gribViewer").off();
}

function reset() {
    $('#gribTimeSelect').empty();
    $("#buttonPath").prop("disabled", true);
    $("#buttonAstar").prop("disabled", true);
    $("#buttonCalc").prop("disabled", true);
    $("#zoom").prop("disabled", true);
    resetGribView();
    resetPathAndAstar();
}

function resetGribView() {
    let canvas = $('#gribViewer').get(0)
    let ctx = canvas.getContext("2d");
    ctx.clearRect(0, 0, ctx.canvas.width, ctx.canvas.height);
}

function addElementsToPath(pos) {
    let lastOfPath = path.pop();
    x0 = lastOfPath["x"];
    y0 = lastOfPath["y"];
    x1 = pos["x"];
    y1 = pos["y"];

    let dx = Math.abs(x1 - x0);
    let dy = Math.abs(y1 - y0);
    let sx = (x0 < x1) ? 1 : -1;
    let sy = (y0 < y1) ? 1 : -1;
    let err = dx - dy;

    while (true) {
        path.push({ "x": parseInt(x0), "y": parseInt(y0) });
        drawPointOnGribView({ "x": x0, "y": y0 }, 1, pathColor);

        if ((x0 === x1) && (y0 === y1)) break;
        let e2 = 2 * err;
        if (e2 > -dy) { err -= dy; x0 += sx; }
        if (e2 < dx) { err += dx; y0 += sy; }
    }
}

function getMousePos(canvas, evt) {
    let rect = canvas.getBoundingClientRect();
    let scale = $("#zoom").val();

    return {
        "x": parseInt((evt.clientX - rect.left) / scale),
        "y": parseInt((evt.clientY - rect.top) / scale)
    };
}

function setGribViewSize(width, height) {
    let scale = $("#zoom").val();

    $("#gribViewer").height(height * scale);
    $("#gribViewer").width(width * scale);
}

function drawGribView(grib) {
    if (grib) {
        let pixels = grib["pixels"];
        let height = grib["height"];
        let width = grib["width"];

        setGribViewSize(width, height);

        let canvas = $('#gribViewer').get(0)
        let ctx = canvas.getContext("2d");

        ctx.canvas.height = height;
        ctx.canvas.width = width;

        let imgData = ctx.getImageData(0, 0, width, height);
        let data = imgData.data;

        for (let y = 0; y < height; y++) {
            for (let x = 0; x < width; x++) {
                let s = 4 * y * width + 4 * x;
                let pixel = pixels[y][x];
                data[s] = pixel[0];
                data[s + 1] = pixel[1];
                data[s + 2] = pixel[2];
                data[s + 3] = 255;
            }
        }
        ctx.putImageData(imgData, 0, 0);
    }
}

function drawPointOnGribView(pos, size, color) {
    let canvas = $('#gribViewer').get(0)
    let ctx = canvas.getContext("2d");
    ctx.beginPath();
    ctx.rect(pos["x"] - (size / 2), pos["y"] - (size / 2), size, size);
    ctx.fillStyle = color;
    ctx.fill();
}

function loadGrib(name) {
    reset();
    $.ajax({
        url: "/api/grib/load/" + name,
        type: 'GET',
        dataType: 'json',
        success: function (grib) {
            currentGrib = grib;
            drawGribView(grib);

            $('#gribTimeSelect').empty()
            let utcunixtimestamp_ms = Date.UTC(grib["year"], grib["month"] - 1,
                grib["day"], grib["hour"], grib["minute"], grib["second"]);

            $.each(grib["times"], function (key, time) {
                $("#gribTimeSelect").append('<option value=' + time + '>' + new
                    Date(utcunixtimestamp_ms + (time * 60 * 1000)).toISOString()
                     + '</option>');
            });

            $("#gribTimeSelect").prop("disabled", false);
            $("#buttonPath").prop("disabled", false);
            $("#buttonAstar").prop("disabled", false);
            $("#zoom").prop("disabled", false);
        }
    });
}

```

```
function getGribs() {
    $("#gribFileSelect").prop("disabled", true);
    $.ajax({
        url: "/api/grib/list",
        type: 'GET',
        dataType: 'json',
        success: function (gribFileNames) {
            $('#gribFileSelect')
                .empty()
                .append('<option selected="selected" value="none"></option>');
            $.each(gribFileNames, function (key, gribFileName) {
                $("#gribFileSelect").append('<option value=' + gribFileName + '>
                    ' + gribFileName + '</option>');
            });
            $("#gribFileSelect").prop("disabled", false);
        }
    });
}
```

A.4 Docker

A.4.1 Dockerfile

```
FROM python:3

WORKDIR /app

RUN apt-get update && apt-get install -y \
  python3 \
  python3-pip \
  libeccodes-dev

RUN pip install --no-cache-dir geopy pygrib matplotlib flask gpxpy

COPY . /app

CMD ["python3", "server.py" ]
```

A.4.2 docker-compose.yml

```
version: "3"
services:
  app:
    build: .
    restart: always
    ports:
      - 5000:5000
```

Appendix B

Attached source code

The full source code of this thesis is embedded into the document. It can be opened with the following link (depends on the pdf viewer).

link to the embedded archive

Suitable document viewer can open such files.

Tested with:

- Firefox 100.0 (`https://www.mozilla.org/en-US/firefox/new/`)
- Okular 21.12.3 (`https://okular.kde.org/`)
- qpdfview 0.4.18 (`https://launchpad.net/qpdfview`)

For printed versions:

The original (digital) version of the document is published under:

`https://doi.org/10.26205/opus-3198`

Note of thanks

Prof. Dr. Sebastian Bab for the support and guidance.

Ulf Müller-Baumgart for the support, correction and annotations.

Samuel Sänger-Böger for the correction and annotations.

Gina Maria Haufe for the correction and annotations.

Udo Cimutta of the Nautical Information Service of the Federal Maritime and Hydrographic Agency Rostock (*Nautischer Informationsdienst des Bundesamt für Seeschifffahrt und Hydrographie*) for the kind support and the provided map samples.

Statutory declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

date, signature

Versicherung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt und mich keiner fremden Hilfe bedient sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Datum, Unterschrift