Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Marine Technology

Ole Brynjar Helland Paulsen

Development of a routing module in a discrete-event simulation platform for ship design assessment

Master's thesis in Marine Technology
Supervisor: Bjørn Egil Asbjørnslett
December 2020

NTNU
Norwegian University of
Science and Technology

Ole Brynjar Helland Paulsen

Development of a routing module in a discrete-event simulation platform for ship design assessment

Master's thesis in Marine Technology
Supervisor: Bjørn Egil Asbjørnslett
December 2020

Norwegian University of Science and Technology
Faculty of Engineering
Department of Marine Technology

NTNU
Norwegian University of
Science and Technology

In memory of Hedvig Kathrine Helland Paulsen

Summary

In recent years, simulation had been investigated as a tool for evaluating ship performances in a realistic environment. The objective of this thesis is creating an algorithm for generating routes between any two ports, and to implement the route-finding algorithm in a simulation model for evaluating vessel performance. Route-finding algorithms that model land areas as polygon obstacles usually depends on generating graphs between the nodes of the obstacle polygons, before the shortest route can be found with a shortest path algorithm. For long-distance routes, a challenge with this approach is the great number of steps needed to both create the graphs and solve the shortest path algorithm. A new heuristic route-finding algorithm is proposed, intended to limit the number of steps needed to find an acceptable route. The algorithm relies on shoreline vector data of GSHHG-format and sailing path polygons describing navigable paths around each shoreline polygon. The heuristic approach involves creating routes around one land area at the time in an iterative manner, while applying a smart selection of which land area to circumvent at each iteration. MATLAB is used for the implementation of the algorithm.

Results are presented for routes between all combinations of the 20 predefined ports, and for routes between a selection of custom defined port locations. The results were evaluated by visual inspection with the objective of feasible approximate shortest path routes. The majority of the routes between the predefined ports, 62.6 %, were evaluated as acceptable. Of the remaining non-acceptable routes, at least 85.9 % are amended by an option to modify the resulting routes. The results illustrate a trade-off between short processing time and the probability of optimal results. A small sample of acceptable routes compared to routes generated by an online service, shows a good match for between the results.

A new simulation model for evaluating vessel performance incorporating the route-finding algorithm, is presented. The simulation model is created with MATLAB and Simulink and is based on a simulation model built for a previous masters thesis by NTNU students. Hindcast met-ocean data is used directly to ensure more realistic rendering of the environmental conditions in the new model, while a Markov chain approach was used in the original model. The results presented confirms the new models ability to give a more realistic representation of the met-ocean conditions. A more realistic model of the met-ocean conditions seems to be gained at the cost of a significant increase in the computational time demand for running a simulation. Reducing the met-ocean data loaded to Simulink could be a beneficial pursuit in that regard. The methods for calculating resistance and estimating speed loss in waves are kept from the original model. The results indicate that the vessel performance parameters with the two models responds in a similar manner, in similar conditions. Evaluating the results from three case studies give valuable information about the model behaviour, but the output performance of the simulation model should ideally be validated against system output data.

Sammendrag

De siste årene har simulering blitt undersøkt som et verktøy for å evaluere ytelsen til skip i realistiske vær- og bølgeforhold. Målet med denne oppgaven er å lage en rute-algoritme som kan implementeres i en simuleringsmodell for evaluering av fartøyets ytelse. Rute-algoritmer som modellerer landområder som polygoner, er vanligvis avhengig av å lage grafer mellom nodene til alle polygonene før standardalgoritmer for å finne korteste vei i en graf kan benyttes. For langdistanseruter er det store antallet trinn som trengs ved å bruke denne tilnærmingen en utfordring, både for å lage grafene og for å løse den korteste rute-algoritmen. Det foreslås en ny heuristisk rute-algoritme som skal begrense antall trinn som er nødvendig for å finne en akseptabel rute. Algoritmen bygger på vektordata av GSHHG-format for å beskrive landområder, og rutepolygoner som beskriver navigerbare ruter rundt hvert landområde. Den heuristiske tilnærmingen innebærer å lage ruter rundt ett landområde av gangen på en iterativ måte, mens man bruker et smart valg av hvilket landområde som skal omgås for hver iterasjon. Algoritmen er implementert i MATLAB. Resultater presenteres for ruter mellom alle kombinasjoner av de 20 forhåndsdefinerte havnene, og for ruter mellom et utvalg av egendefinerte havner. Resultatene ble evaluert ved visuell inspeksjon med seilbare tilnærmet korteste ruter som mål. Flertallet av rutene mellom de forhåndsdefinerte havnene, 62,6 %, ble vurdert som akseptable. Av de gjenværende ikke-akseptable rutene, kan minst 85,9 % av rutene utbedres ved å bruke det inkluderte metoden for å utbedre resulterende ruter. Resultatene illustrerer en avveining mellom kort beregningstid og sannsynligheten for gunstige resultater. Et lite utvalg av akseptable ruter, sammenlignet med ruter generert med en onlinetjeneste, viser godt samsvar mellom resultatene.

En ny simuleringsmodell for evaluering av fartøyets ytelse, som inkluderer rutealgoritmen, presenteres. Simuleringsmodellen er laget med MATLAB og Simulink, og er basert på en simuleringsmodell bygget for en tidligere masteroppgave av NTNU-studenter. Hindcast værdata brukes direkte for å sikre mer realistisk gjengivelse av miljøforholdene i den nye modellen, mens en Markov-kjede tilnærming ble brukt i den opprinnelige modellen. Resultatene som presenteres bekrefter den nye modellens evne til å gi en mer realistisk fremstilling av værforholdene. En mer realistisk modell av værforholdene ser ut til å bli oppnådd på bekostning av en betydelig økning i beregningstiden for å kjøre en simulering. Å redusere mengden værdata som er lastet til Simulink, kan være en gunstig forfølgelse i den forbindelse. Metodene for å beregne motstand og estimere hastighetstap i bølger er beholdt fra den opprinnelige modellen. Resultatene indikerer at parametere for fartøyytelse med de to modellene reagerer på samme måte under lignende værforhold. Evaluering av resultatene fra tre casestudier gir verdifull informasjon om modellens atferd, men simuleringsmodellen bør valideres mot systemdata.

Preface

At the start of the spring semester of 2009 I was halfway through my studies at NTNU. I was looking forward with delight to finishing the last half of my masters degree. The course topics was getting increasingly interesting and I had started to get friendly with more of my fellow students. On February 14 in 2009, life took a strange turn and things did not turn out as expected. A blow to the head during a basketball game resulted in visual disturbances, a constant feeling of dizziness and an increasingly severe fatigue. As I am writing this almost twelve years later, my condition has not changed, despite numerous attempts at treatments. Had I known back then that my assumption of a gradual recovery would not pan out, I would probably not have kept on working to finish my studies through all these years.

The period working on this masters thesis started in August of 2017 and has been eventful and challenging. On November 18, 2017 my life changed in a wonderful way as my son was born. As amazing and delightful as he is, with a new top responsibility and my reduced condition, less energy has been available for working on the thesis. Then in November of 2018 my sister Hedvig tragically passed away after battling cancer for almost two years. A devastating loss that have impacted my priories for periods of time and impacted my ability to focus at other occasions. Hedvig always implored me to be rational and keep on working. I miss her dearly. Finally, the year 2020 has been a strange year for all of us with a global pandemic and restrictions imposed on society as a response. The period with closed kindergarten and university facilities were not exactly helpful in my quest to complete my masters degree. With all those challenges in mind it is with great satisfaction, some relief and bit of pride, that I am finally submitting my masters thesis.

I would like to thank both the Department of Marine Technology and the Faculty of Engineering at NTNU, for being flexible in lettering me pursuit my studies through all these years. In the same regard I would like to extend my gratitude to my thesis supervisor Bjørn Egil Asbjørnslett, I greatly appreciate the patience and flexibility. Gisle Marhaug at NTNU has also been a great support for me since restarting my studies in the autumn of 2016. We had weekly talks for a long period of time, and I am truly grateful for all the help given to me. Lastly, I must mention my thesis co-supervisor Endre Sandvik, who has guided my work on this thesis for about three years. I have received far more guidance hours than should be required of any supervisor, and for that I am tremendously grateful.

Ole Brynjar Helland Paulsen

Trondheim, December 21, 2020

Table of Contents

List of Tables

List of Figures

Nomenclature

BN	=	Beaufort wind scale number
ECA	=	Emission control area
ECMWF	=	European Centre for Medium-Range Weather Forecasts
GSHHG	=	Global Self-consistent, Hierarchical, High-resolution Geography Database
nm	=	Nautical miles
V	=	Vessel speed
H_S	=	Significant wave height
Mwd	=	Mean wave direction
T_P	=	Peak wave period
$v10$	=	Northward windspeed component
$u10$	=	Eastward windspeed component
U_{res}	=	Resultant wind speed
θ_{rel}	=	Wind speed component in direction of the ship heading
U_{rel}	=	Wind speed component in direction of heading relative to the vessel speed
C_{air}	=	Air density
A_p	=	Transverse projected area of the superstructure
R_{AA}	=	Resistance due to air and wind

Chapter 1

Introduction

In recent decades the world has seen an increased focus on climate and work on solutions to restrict greenhouse gas emissions. The last decade has also revealed volatile fuel prices. As a result of the demands for reduced emissions combined with motivation for reduced operational costs, the maritime industry has now keen focus on fuel efficiency and low emission technologies. This creates an opening for innovative and energy-efficient ship designs. The traditional approach to this is however conservative, and new designs are often based on existing vessels and proven technology. Ships are commonly built to comply with the ship-owners requirements for attainable design speed and fuel consumption at given speeds. Traditionally these requirements have been met by designing ships for calm water conditions and then adding a sea margin of $15 - 30$ % (Arribas, 2007).

Simulation technology have been explored as an aide to ship design in recent years. There are several reasons why simulation technology can prove to be a valuable tool in the design process. A reliable and realistic simulation model would ideally allow for testing different design solutions in an inexpensive way, and thus enable the comparison of a wider array of feasible solutions at an early design stage. Furthermore, ship owners may be more likely to implement novel technology in a new design, if the ship performance can be validated by a trusted simulation model. This is relevant for the Norwegian maritime industry, since many ships built in Norway are highly specialised and costly vessels with many customised design solutions.

Simulation also allows for testing the performance of a ship design in a realistic environment, by looking at the specific weather conditions for the area where a ship is intended to operate. Potentially allowing for more precise calculations of attainable speed and necessary power, compared with the traditional methods. In addition to have the ability to test the performance of different designs in a given environment, it is also of interest to be able to test the performance of a given design in different environments. In addition, also possibly highlighting the strengths and weaknesses of the vessel design. Such capability could also enable finding the most likely optimal route between two ports for a given vessel and season. An accurate simulation model, with the ability to simulate the sailing along different routes and with a realistic representation of the environmental conditions

along each route, could potentially allow companies to optimise both the route and vessel design for maximum revenue.

For the mentioned purposes, a simulation model should have the capability to simulate sailing along several different routes, ideally without having to alter the model. A simulation model could therefore be enhanced by including a feature that allows for creating routes automatically between any two ports, eliminating the need to process the route data in advance.

1.1 Background

In the spring semester of 2017, two graduate students at NTNU created a simulation model for their master‘s thesis. The purpose of the model was to analyse the performance of a vessel at the early design stage, in a more precise manner than commonly done by designers at the time(Bakke and Tenfjord, 2017). The thesis, named *Simulation-Based Analysis of Vessel Performance During Sailing*, focused on getting an accurate calculation of the added resistance due to waves and wind. Bakke and Tenfjord (2017) presents a case for a route across the Pacific Ocean, where the route is given as manual input of the coordinates of seven waypoints. Their model does not have the ability to set up or change routes, in order to simulate and evaluate the vessel performance in different environments. Motivated by the previously stated reasons, it is of interest to add such capability.

The shortest route between any two points on the globe is a great circle between the two locations. For aviation purposes this could be a final route. For shipping purposes however, the problem complexity is far greater, simply because ships are confined to sail on water. A great circle route between two ports will commonly intersect several land areas. The fundamental problem is thus to find the near shortest route between any two ports, that is navigable, and that omits all land areas. Furthermore, to create a model to automatically create such route, and to incorporate this functionality in a simulation model. With the objective to enable simulating the performance of a vessel sailing along routes between any two ports, while accounting for the specific weather conditions encountered along the given route. This the problem addressed in this thesis.

1.2 Previous work

A review of literature relevant to the thesis objective is presented following. A brief presentation of relevant literature regarding general shortest path problems, and the specific problem of route-finding algorithms in the maritime domain, is first described. This is then followed by a brief summary of relevant literature regarding simulation applied to ship design.

1.2.1 Route-finding algorithms

A multitude of papers have been published regarding finding the shortest or minimum cost paths in a graph. Dijkstra (1959) propose an algorithm for finding the minimal distance path from a given node to all other nodes in a graph. The algorithm presupposes at least

one arc connecting each set of nodes, and that each arc have a fixed weight or distance. The method can be used on both directed and undirected graphs. Several alternatives to, and modifications of the algorithm has also been published. Bellman (1958), Johnson (1977), Bovet (1986) and Hart et al. (1968) all presents algorithms for finding the shortest path in an weighted graph. Möhring et al. (2006) presents four methods of partitioning a graph in order to speed up Dijkstra's algorithm. Partitioning involves dividing a graph into regions in order to limit the search space. The partitioning methods are combined with a method involving using pre-calculated variables indicating if an arc is a starting point for a shortest path to a given region of the partitioned graph.

Rohnert (1986) and later Fagerholt et al. (2000) present methods for finding the shortest path between two points in the presence of polygon obstacles by first establishing a partial visibility graph, and then applying Dijkstras method. The method by Fagerholt et al. (2000) is specifically for finding the shortest path from a ships position to a destination port in the presence of polygon obstacles, that are defined such that it is possible to sail along the outer edges of the polygons. The ship node is not a part of the partial visibility graph called an operating network, while the destination port is defined to be a node in one of the obstacle polygons.

Chew (1986) presents a solution for finding an approximate shortest path between to point in the presence of polygon obstacles, when the points of origin and destination are vertices of the polygons. Dijkstras algorithm is solved for what is described as a Delaunay-like graph, called an obstacle triangulation. It is to be constructed in less time than the visibility graph. Beeker (2004) describes a method for finding the shortest path that involves dividing the navigable waterways into areas in such a way that the next point on the shortest path to a destination, is equal for all points within each area. The algorithm utilises a Delaunay triangulation which connects the nearest neighbours in a set of points.

Tanaka and Kobayashi (2017) presents an algorithm for finding the optimal route for minimum fuel consumption, when accounting for drift due to currents. The problem is solved by combining an algorithm for optimal speed along a fixed path, with an algorithm for finding the optimal route for a fixed sailing speed. The article presents the results of experiments for an ocean area between Japan and Taiwan. The graph nodes are equally spaced in a grid pattern. Since all nodes are in open water, the utility for generating routes from port to port is not demonstrated.

Xu et al. (2012) describes an algorithm for providing the shortest path and the least turning points. The method is based on generating raster data from vector electronic charts. A buffer zone is added to the rasterised obstacles. Each obstacle is given a colour value which is used to identify whether a route between two points intersects with an obstacle. The method presented first creates a feasible route from origin to destination, then a rubber band algorithm is implemented to improve the route to ensure the shortest path. Another algorithm based on raster charts in presented by Chang et al. (2003), in a paper introducing a maze routing algorithm for finding optimal routes with collision avoidance. The algorithm can find the shortest path between a pair of cells on a rectangular grid of N cells in N steps.

1.2.2 Simulation applied in ship design

Chen et al. (2011) presents a model, utilising simulation for optimising vessel design parameters and number of vessels in a fleet, with the objective of minimising the logistics cost per ton iron ore. Monte Carlo simulation is utilised, where voyage duration, waiting time at port, and weekly iron ore demand are determined based on probability distributions. The simulation output is logistics cost per ton for given set of design parameters, number of vessels, reorder level and initial inventory. The model does not give a detailed evaluation of the performance of a given design in specific environmental conditions.

Kauczynski (2012) presents a method for evaluating vessel performance by means of numerical simulation, applying hindcast weather data. The paper stated that applying complete historical weather time series along a considered route, is the most reliable method for studying reliability of the ship transportation. And that using long term statistics to represent weather conditions by means probabilities, results in an incorrect representation of the extreme value tail of the probability distributions of sailing time. The presented methodology is first calculating the ship performance for sets of different wind, wave and swell parameters. Then generate a set of matrices with coefficients expressing speed loss, delivered power and fuel consumption based on the oceanic and weather conditions. The matrices are used as input in the simulation model. Where the performance parameters at each simulation update are calculated by interpolation based on the given environmental conditions at the specific location. The simulation model is used to generate probability distributions for voyage duration for different seasons. According to the paper, this method results in a good quantitative estimation of the mean sailing time.

Fathi et al. (2013) presents a benchmarking methodology for evaluating ship designs through simulation of the ship‘s operation over several years. The model incorporates hind-cast weather data and a detailed mission profile in order to simulate realistic operation conditions for the vessel. The advantage of this methodology is that it provides detailed operational profiles of ship designs, and thus enables a realistic comparison between different designs at the early design stage. The simulation model consists of a model of the ship, a logistics model and met-ocean data. The methodology presented utilises the software package ShipX for calculations of the hydrodynamic characteristics of a given design. In the logistics model, the ships operation is defined by a set of routes. The sailing legs between two ports are divided into waypoints in order to update the simulation model with the corresponding conditions at each step. The model calculates the total resistance in seaways at each waypoint, including wind resistance. The model also includes cargo handling at port and adjust the ships displacement in accordance with the amount of cargo unloaded or loaded.

Erikstad et al. (2015) presents a workbench for performing a “virtual sea trial”, intended to aid in the ship design process and to verify and document vessel performance. Capturing complex interactions of hydrodynamics, power production and service equipment, would allow for benchmarking of energy efficiency and emissions to air of alternative designs. The model is presented as able to simulate both long-term performance and specific short-term operations. The paper points out the challenge of comparing real life performance of different designs, due to differences of operating conditions and missions, between two similar trips. And further, that simulations over the lifetime of a vessel can yield detailed realistic operation profiles. A more realistic comparison of vessel de-

signs can then be achieved by long-term simulation of vessel performance in its intended operation conditions.

1.3 Scope

The objective of this thesis is two-fold. The first part is to create a model for automatically generating feasible routes for sailing between any two ports. The second part is to implement the route-finding algorithm in a simulation model, with the end goal to enable simulating a vessel sailing between any two ports in a simple manner.

The challenge addressed by the first aim, is not to find common routes between the busiest ports worldwide. Such routes could be generated from available data, for instance by tracking AIS data from websites offering such services. It would then be trivial to store a given number of routes between a fixed set of ports. The problem this thesis intends to solve, is to create an algorithm implementable in MATLAB that can set up feasible and realistic routes between any two points on the globe. The route-finding model should be versatile and be able to create routes between both pre-defined and customised ports. An important aspect is to create a MATLAB program that ensures this feature, enabling the model to easily be implemented in the simulation model. The route-generation model should add value to the simulation model by ensuring the ability to compare ship performance on several different routes, without having to manually establish the routes in advance. The final product should be a robust algorithm for automatic route-generation, able to generate routes between any two ports worldwide, without the need for pre-processing of any kind. A database of ports to be chosen from should be included as a supplement to the ability of defining custom ports.

The second aim of this thesis is to present an updated simulation model, simulating a vessel sailing between two ports, and evaluating the vessel performance and to incorporate the algorithm for automatic route generation in the simulation model. The new simulation model should be based on the simulation model presented by Bakke and Tenfjord (2017). Specifically, the methods for evaluating vessel performance should be carried forward in the new model. In the original model, the weather data for each sailing leg was loaded from separate files. Having to create such files for each sailing leg defeats the purpose of enabling simple and automatic route generation, since the goal is to minimise the need for prepossessing. The new simulation model should use historic weather conditions directly, instead of stochastic methods. The weather data should be loaded from a single file for any route. Also, the original model only incorporates time as fixed time steps between simulation updates. In order to use historical weather, time and date must be incorporated as simulation variables. The final product should be a simulation model able to simulate sailing between any two ports, without requirements of any manual processing of route or weather data.

1.3.1 Limitations

In real life several considerations besides the shortest path, may determine the route a ship will sail. Examples of reasons to deviate from the shortest path can be weather and climate conditions, currents, limitations in canals and pirate activity. Determining the optimal

route for a given vessel requires knowledge regarding specific environmental factors in the current waters to be navigated. And such knowledge is likely best achieved from years of experience sailing the same waters. Specific environmental knowledge is also hard to incorporate in a versatile and generic model. For these reasons, the route-finding model will only consider the shortest distance when determining the best route. The result should, however, be a feasible route circumventing all land areas.

The custom ports are restricted to be located in coastal areas, with open access to deep sea sailing. The route-finding model should be created in MATLAB, and existing software for automatic route generation will therefore not be considered. Geographical vector data for the complete globe is freely available online in a format easy to implement i MATLAB. The algorithm for automatic route-finding shall therefore be based on geographical vector data, and raster data methods will not be considered.

The purpose of the new simulation model is not to improve the methods of predicting resistance, power consumption or speed loss that is incorporated in the original model. It is rather to demonstrate that the route-finding capability can be implemented in a simulation model. The methods for performance calculations are thus kept from original model. The existing model have two modules that can be used for resistance calculation, one which utilise ShipX, and one which use the empirical method of Hollenbach. Only the latter will be included in the new model since it relies on fewer weather parameters.

For the purpose of evaluating the vessel performance in a realistic environment, the important part of the ship operation is the sailing between harbours. The time spent in harbour is of little value when evaluating the ship performance in different weather conditions. Evaluating operational profiles are beyond the scope of this model and therefore only one-way trips between two ports will be evaluated. Time spent in harbour is not incorporated. Reduced speeds near shore, time slots in canals, or reduces speed or alternative power sources in emission control areas, are all aspects that could be included to create a more realistic simulation model. New aspects are not considered, besides incorporating hindcast weather data, time and date, and the ability to generate new routes.

1.4 Thesis structure

The remainder of this thesis is organised as described in the following. Some relevant background theory for the route-finding algorithm, including motivation for the chosen methodology, is presented in chapter 2. While the details of the model for automatic route generation is described in chapter 3. Chapter 4 presents a brief introduction to simulation theory, and some relevant background for the simulation model. Which itself is detailed in chapter 5. Results for both the route-generator and the simulation model is revealed in chapter 6, followed by a discussion in chapter 7. Lastly, chapter 8 presents some conclusions and recommendations for further work.

Chapter 2

Background for the route-finding algorithm

This chapter presents theory and concepts relevant for creating a route finding algorithm, and lays out the methodology for the route-finding model presented in chapter 3.

2.1 Background theory

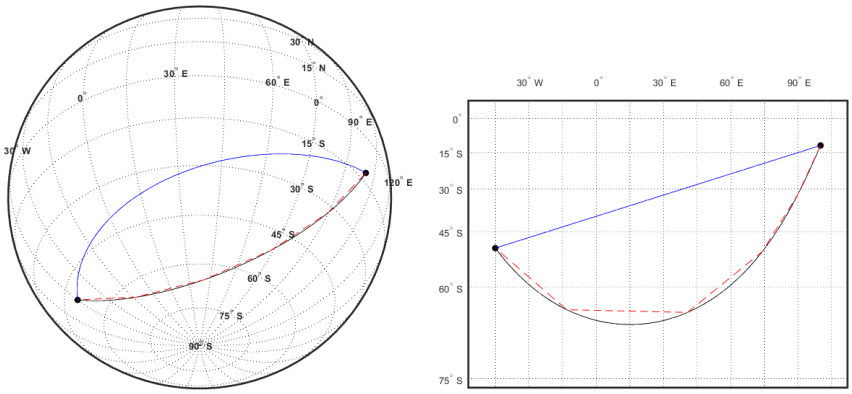
Some relevant concepts and a few shortest path algorithms are briefly presented in the following.

2.1.1 Relevant concepts

Great circles and rhumb lines

The shortest distance between two points on a sphere is found by a great circle. More precisely defined, a great circle is the line of intersection between a sphere and a plane passing through the centre of the sphere(MathWorks, 2019a). The equator and the meridians are great circles that follow a constant course. All other great circles will intersect the meridians at different angles. Lines intersecting the meridians at a constant angle, are called rhumb lines(MathWorks, 2019a). Both a rhumb line and a great circle between two point on a globe, is illustrated in figure 2.1. The rhumb line is displayed in blue colour and the great circle in black.

Sailing along a great circle requires constantly changing the heading. It is thus much easier to navigable along a rhumb line path. An approach for sailing an approximate great circle route without frequently adjusting the heading, is to divide a great circle route into waypoints and sail along rhumb lines between each waypoint. An approximate great circle route is displayed by the dotted red in in figure 2.1.



(a) Orthogonal projected map **(b)** Mercator projected map

Figure 2.1: Great circle and rhumb line between two points on a globe.

Convex hull

In two dimensions, a subset S is convex iff all line segments between any points p and q are fully contained in S. The convex hull CH(S) of a subset S, is the smallest convex set that contains S(Li and Klette, 2011). A convex hull around a set of nodes N, is thus a polygon made up of line segments between the outer nodes N, in such a way that none of the nodes are located outside the convex hull. An example of a convex hull is illustrated in figure 2.2.

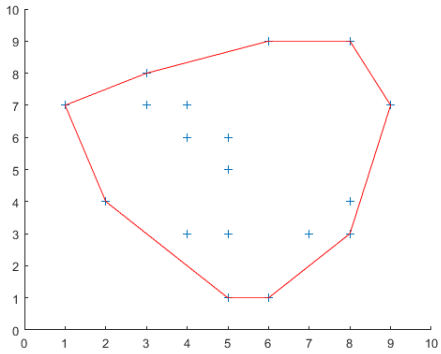
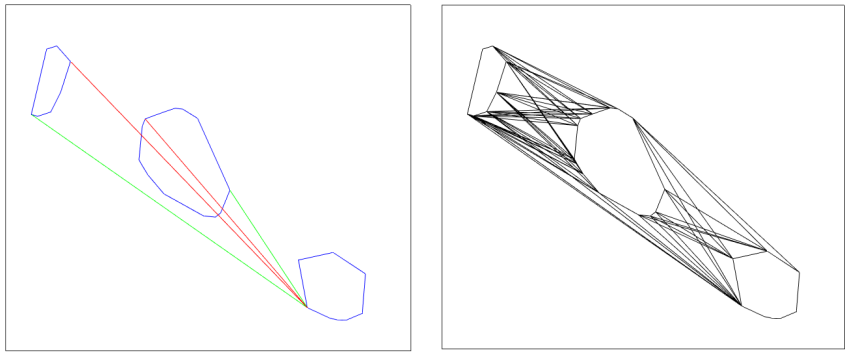


Figure 2.2: Example of a convex hull

Visibility

The concept of visibility can be defined in different ways for different problems. Two nodes can be said to be *visible* to each other if it is possible to travel along an arc from one

node to the other without intersecting an obstacle. For the work in this thesis, polygons are considered as obstacles, and visibility is mostly investigated by means of great circle routes. Visibility by rhumb lines are however also considered in some cases. Figure 2.3a illustrates the concept of visibility, where the green lines depict arcs between nodes that are visible to each other and the red lines show arc between nodes that are not visible to each other.



(a) Visible and non-visible pair of nodes **(b)** Visibility graph

Figure 2.3: Visibility between the nodes of polygons.

Graphs

Graphs are typically made up of a set of nodes and a set of arcs connecting at least one pair of nodes each. Commonly arcs have a cost associated with including it in a path. For real world problems such cost is a representation of the distance travelled. Some graphs are bidirectional, which mean that the cost of traversing an arc can depend on the direction travelled. A *visibility graph* is graph where arcs are made only between nodes that are visible to each other. An example of a visibility graph for the nodes of three polygons is illustrated in figure 2.3b.

2.1.2 Shortest path on a graph

According to Möhring et al. (2006), Dijkstra's algorithm has become the standard for solving the shortest path problem on a graph. Time complexity is a term used for describing how many steps it takes to solve a problem, while space complexity is a description of the amount of memory needed to solve the problem. If the shortest path in a graph with n vertices can be solved in n^2 steps, it is common to describe the time complexity as $O(n^2)$(Chang et al., 2003). The shortest path between two points in a graph with V vertices can be solved in $O(V^2)$ time by applying Dijkstra's algorithm(Chew, 1986). Several heuristics for speeding up the solution process has since been presented. Two such methods are described below.

The *A-star* algorithm by Hart et al. (1968) incorporates a heuristic cost function, estimating the cost of an optimal path from a given node to the goal node. The heuristic cost

function depends on special knowledge regarding the specific problem and is used as a lower bound for the minimal cost solution. In maritime domain such lower bound could equal the great circle distance, since the length of any route between two locations cannot be less than length of great circle route. An evaluation function sums up the estimated cost of an optimal path from the start node to the current node, and the estimated optimal path cost from the current node to the end node. Where the former is estimated by the shortest path to the given node found thus far, and the latter is the value of the heuristic cost function for the given node. The evaluation function is used to select which node to investigate next. Unlike Dijkstra's algorithm, where all nodes are investigated, the A-star algorithm seeks to investigate as few nodes as possible. The success depends on the fitness of the heuristic cost function. If the heuristic cost function has a value of zero for all nodes, the algorithm becomes equal to Dijkstra's. If heuristic cost function does not represent a proper lower bound for the cost of the optimal path from a node to the goal node, an optimal path may not be found.

The focus of many developed shortest paths algorithms has been to achieve a low theoretical time bound, which does not necessarily correspond to low computational time. According to Möhring et al. (2006), such algorithms are often too slow for application in large networks that requires many shortest path computations. The partitioning methods presented by Möhring et al. (2006) use arc-flags, logical variables indicating if a given arc is a starting point for a shortest path to a given region of the partitioned graph. The arc-flags are stored in one vector for each arc, with one entry for each region of the partitioned graph. Each entry represents a specific region. Both unidirectional and bidirectional search with Dijkstra's algorithm is combined with the different methods of partitioning. In a bidirectional search, two Dijkstra's algorithms run simultaneously from both the start node towards the target, and in the opposite direction with the reversed graph. The search alternate between the two directions until the two paths meet. Combinations of bidirectional search with two of the partitioning methods yielded speeds over 500 times faster than the ordinary application of Dijkstra's algorithm, when applied on a German road networks.

Shortest path in presence of polygon obstacles

A challenge when using shortest path algorithm for maritime applications, is that there is not a pre-defined fixed network for shipping routes. The oceans are wide and the possible routes a ship can sail across the open sea cannot be easily quantified. A substantial part of the problem is therefore to define a graph prior to applying a shortest path algorithm. The methods by Rohnert (1986) and Fagerholt et al. (2000) offers possible solutions, as both present methods of creating partial visibility graphs between polygon obstacles.

In the method presented by Rohnert (1986), the partial visibility graph is made up of the line segments of the polygons and supporting segments, where the latter is defined as lines between vertices of two polygons that are common tangents to both polygons. The author states that a line segment between two nodes of two polygons cannot be part of the shortest path if it is not a supporting segment, and that there exists at most four supporting segments between two disjoint convex polygons. Supporting segments between two polygons that intersect other polygons are not a part of the solution. For a problem with f disjoint convex polygons with n vertices in total, the partial visibility graph consists of the n line segments of the polygons plus at most $4f$ supporting segments from the

endpoints to the polygons, plus at most $4f^2$ supporting segments between the polygons. The partial visibility graph can be computed with a time complexity of $O(n + f^2 \log n)$. An the shortest path can be found by Dijkstras algorithm in $O(f^2 + n \log n)$ time.

The method presented by Fagerholt et al. (2000) generates an operating network, which partially corresponds to the visibility graph from the line segments of the obstacle polygons. The method defines outer nodes, and port entrance nodes, where the latter are thought to be connected to internal ports through local networks. The operating network consist of arcs between all mutual outer nodes, and between port entrance nodes and outer nodes. Great circle lines are approximated by straight lines, justified by the limited distances. In the article two shortest path algorithms are applied, both based on Dijkstra's algorithm, as the method is stated to be suitable for short sea voyages with sailing times up to a few days. The efficiency of the algorithm is highlighted by comparing the results to an algorithm that utilises the complete visibility graph.

2.2 Model background and methodology

The problem in question is to present a route-finding algorithm capable of creating routes between any two ports worldwide. The proposed solution utilise the idea of obstacle polygon defined in such a way that it is possible to sail along the arcs of the polygon. A total of 5863 polygons are used, with the number of nodes in each polygon ranging from 3 to 168. In order to use a shortest path algorithm, it is necessary to create a partial or complete visibility graph with all the relevant polygon obstacles. The number of polygons involved in a problem depends on the span between the port locations. For ports located on opposite parts of the globe, a great number of polygons and total amount of nodes will be included in the problem. Creating a partial visibility graph in such cases will require a tremendous number of steps to compute. The same is true for running Dijkstra's algorithm on such network. Even if the time complexity can be improved by the presented method for speeding up the algorithm, they both requires a fair degree of prepossessing. Möhring et al. (2006) notes that there seems to be a trade-off between memory usage and the speedup factor achieved.

A heuristic algorithm is proposed in order to reduce the number of steps needed to create a route. To achieve this, the algorithm intersects one land area at the time and incorporates a smart selection of the order of land area to circumvent. This heuristic is inspired by the cost heuristic of the A-star algorithm inn the way that one wishes to expand as few as possible nodes, and only those which is likely to be included in the minimum cost path. It is desirable to inspect and possibly circumventing as few land areas as necessary. Routes around each polygon is created by first finding suitable nodes to connect a port or node of previous polygon to the intersected polygon. The idea is to expand as few arcs as possible. A fundamental part is to utilise convex hull in this task. A convex hull around the two endpoints and the nodes of the polygon obstacle, gives the shortest way circumvent all the included nodes, in a plane. It can thus also be used to find the two shortest paths between the endpoints, around the polygon obstacle. While giving an exact solution for a problem in the plane, it is assumed that using convex hulls for the presented problem, gives a good approximation for problems involving geographical coordinates. The proposed solution is partly selected based on functionality available in MATLAB.

2.2.1 Software

The code for the route-finding algorithm is written in MATLAB, a programming platform and programming language used by many engineers and scientists. The MATLAB programming language relies on libraries for matrix manipulation, and a multitude of MATLAB functions. Several toolboxes can be added to expand the programming capabilities.

MATLAB code can be written and stored in scripts and functions. When scripts are executed, the variables defined within the code is written to the MATLAB workspace. A function can read declared input variables, and only return variables as specified.

A MATLAB structure array is a useful variable for storing and handling data. Each element of a structure array can contain several fields of information of different data types. A given element of a structure array can thus describe a variable by attributes of different format, such as character strings, vectors, matrices or scalar numbersMathWorks (2019b).

Mapping toolbox

The Mapping Toolbox is a toolbox added to MATLAB, used for analysing, manipulating and displaying geographic data. Both vector and raster data can be imported, and a wide range of file formats are supportedMathWorks (2019a). The toolbox enables working with geographical data structures. These are structure arrays containing one element for each geographical structure.

2.2.2 GSHHG data

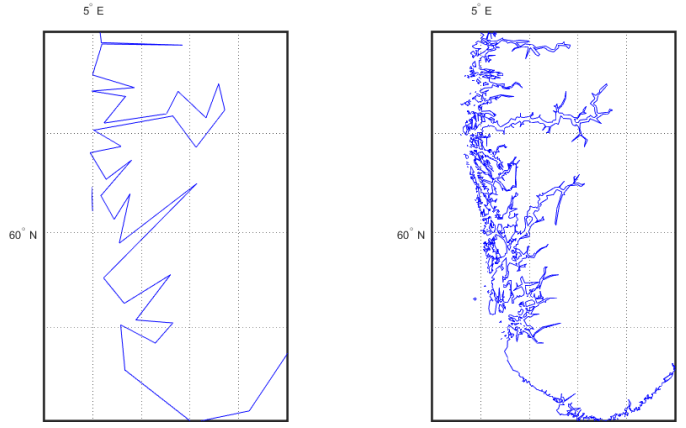
The Global Self-consistent, Hierarchical, High-resolution Geography (GSHHG) Database is used to model the world geography. The freely available GSHHG dataset describes geographical boundaries as polygons, whose nodes are described by latitude and longitude vectors. The data is divided in six levels describing different geographical boundaries, listed in table 2.1(Wessel and Smith, 2019).

Table 2.1: Geographical boundaries described by each level of the GSHHG data(Wessel and Smith, 2019)

Level	Geographical boundary type
1	Between land and ocean, except Antarctica
2	Between lake and land
3	Between island-in-lake and lake
4	Between pond-in-island and island
5	Between Antarctica ice and ocean
6	Between Antarctica grounding-line and ocean

When read in MATLAB, the data is loaded to a geographic data structure variable, containing one element for each polygon. Each element contains 21 fields of information about each geographical shape. The latitude and longitude data are two such fields. Other fields describe extreme coordinates of the polygon, the level type, a unique polygon ID,

and if the polygon goes across the international date line(MathWorks, 2019c). A list of all fields is included in appendix B.



(a) Polygons of crude resolution **(b)** Polygons of intermediate resolution

Figure 2.4: Map of the west coast of Norway from shoreline polygons of different resolution

The GSHHG data is available in full, high, intermediate, low and crude resolution. Each step down from the full resolution yields a reduction in size and quality of about 80 percent(Wessel and Smith, 2019). Table 2.2 gives an indication of the amount of data included in the different resolutions. The difference is further illustrated in figure 2.4, with a juxtaposition of a map from crude resolution data, with one from intermediate resolution data.

Table 2.2: Data included in the different resolutions of the GSHHG data

Resoution	Polygons included	Level 1 polygons	Nodes in Eurasia polygon
Full	188612	179832	1181110
High	153462	144744	141705
Intermediate	41230	32825	34832
Low	10717	5701	6851
Crude	1781	731	1028

Chapter 3

Model for automatic route-generation

The basic idea behind the model for automatic route generation, is to have predefined sailing paths around all land areas. The paths are described by polygons divided in straight legs of variable lengths, separated by nodes, henceforth called *shoreline sailing path polygons*. When a great circle route intersects a land area, a suitable route is found by identifying optimal *connection nodes*, where an open seas route is connected to a shoreline sailing path polygon. The concept is illustrated in figure 3.1. The open seas parts of the route is shown as a blue line, while the shoreline path is coloured green.

When several land areas are intersected, the method is used in an iterative process until a route omitting all land areas is found. A near shortest path can be found by smart selection of the sequence of land areas to omit.

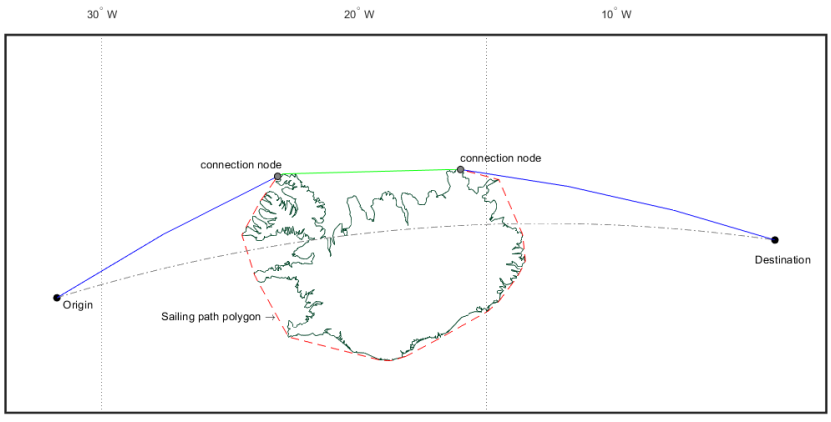
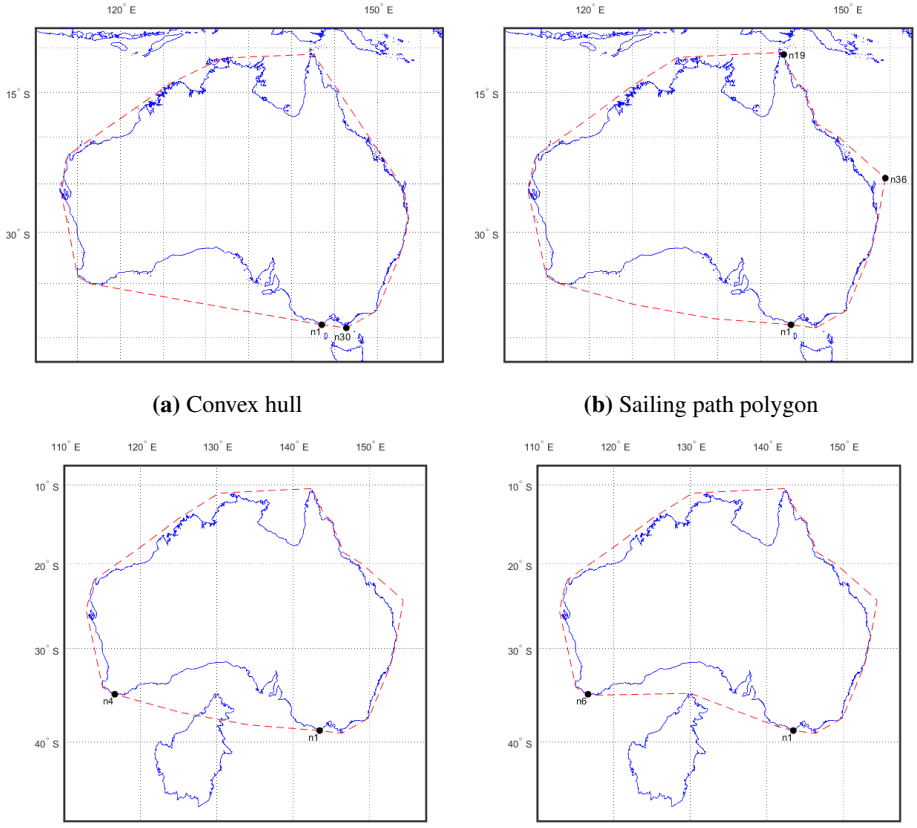


Figure 3.1: Basic concept of finding a route circumventing a land area.

The remainder of this chapter is devoted to describing the model for automatically generating routes between two ports. First the input data is described, then an overview from a user perspective is presented. The route-finding algorithm is then described in two steps. First the method of creating a route circumventing a single land area is detailed in section 3.3, then the process of circumventing several land areas are described in section 3.4.

3.1 Input data

The model relies on two types of input data, loaded from three pre-stored files. Two files with GSHHG data and one file with data describing shoreline sailing path polygons. An example of the latter is illustrated in figure 3.2. Both types of input data is described in the following.



(a) Convex hull

(b) Sailing path polygon

(c) Hull intersect another shoreline polygon

(d) Modified sailing path polygon

Figure 3.2: Shoreline sailing path polygons describe navigable paths along the shorelines of land areas.

3.1.1 Shoreline vector data

Two sets of shoreline data are used, one for visual display of routes on maps, and another for use in the route-finding algorithm. A customised dataset is used for the latter purpose, while intermediate resolution GSHHG data is used for the former. The intermediate resolution data is stored in MATLAB-file format.

The custom GSHHG data is used to check if a route between two points intersects any land areas. For this use, it is not reasonable to include other levels such as lakes, and rivers. Also, the ice sheet, rather than the Antarctic continent seems more fitting as a boundary for maritime traffic. Thus, only the land areas and the Antarctic ice sheet, described by level one and five polygons, are included.

Intermediate resolution data is used for the custom data set as well. But for the level-one polygons, the data set only includes the polygons included in the low-resolution data. This is done to limit the processing time, by restricting the number of polygons to search through for intersections. 5862 polygons are included in total. The data is stored in shapefile format and loaded to MATLAB as a geographical structure array.

3.1.2 Shoreline sailing paths

The model utilises pre-defined shoreline sailing paths around land areas, in order to find the shortest path circumventing a land area. The sailing paths are polygons made from convex hulls, created around all the shoreline polygons in the custom GSHHG dataset. A convex hull is made up from straight lines connecting the outer nodes of a set of nodes, in such a way that no other nodes lies outside the convex hull. Figure 3.2a shows a convex hull around the Australian coastline.

The convex hulls are in turn altered to make the paths feasible sailing routes. This can be done for several reasons. In some cases, the distance between two nodes is so great that the straight line between the nodes of the convex hull is a poor approximation of a great circle route between the nodes. In such cases nodes are added, so that the route between the two nodes are divided in sailing legs, approximating a great circle route. This is illustrated in figure 3.2b, where the sailing path south of Australia is altered compared to the convex hull of figure 3.2a. The sailing path is also altered along the north east coast between nodes 19 and 36 in figure 3.2b. Here the path is set to a navigable route inside the Great Barrier Reef. Underwater reefs is not described in the shoreline data, and must be accounted for manually to ensure a feasible route.

Parts of some convex hulls will intersect other land areas. A fictive example of this is shown in figure 3.2c, where an island has been inserted between nodes 1 and 4. In such cases the hulls are modified such that the path between the two nodes is the shortest path around the intersecting land area. This is illustrated in figure 3.2d. Canals are a final example of areas where the shoreline hulls must be modified extensively in order to describe a navigable route. The hulls have been modified to follow the great canals in an approximate manner.

The data set of shoreline sailing routes is stored in shapefile format, and loaded to MATLAB as a structure array, containing 5862 element. Each element is described by ten fields of information. A custom field is added to indicate if a given element is a convex hull or if has been modified. Given that the convex hulls must be modified by visual inspection

to ensure navigable routes, it would be an overwhelmingly demanding task to modify all polygons in the data set. Convex hulls have been generated around all the included shoreline polygons, but only 28 of the convex hulls have been modified to ensure feasible sailing paths. The largest land areas, and areas located in important shipping lanes have been prioritised. A list of which shoreline sailing path have been modified is included in appendix C.

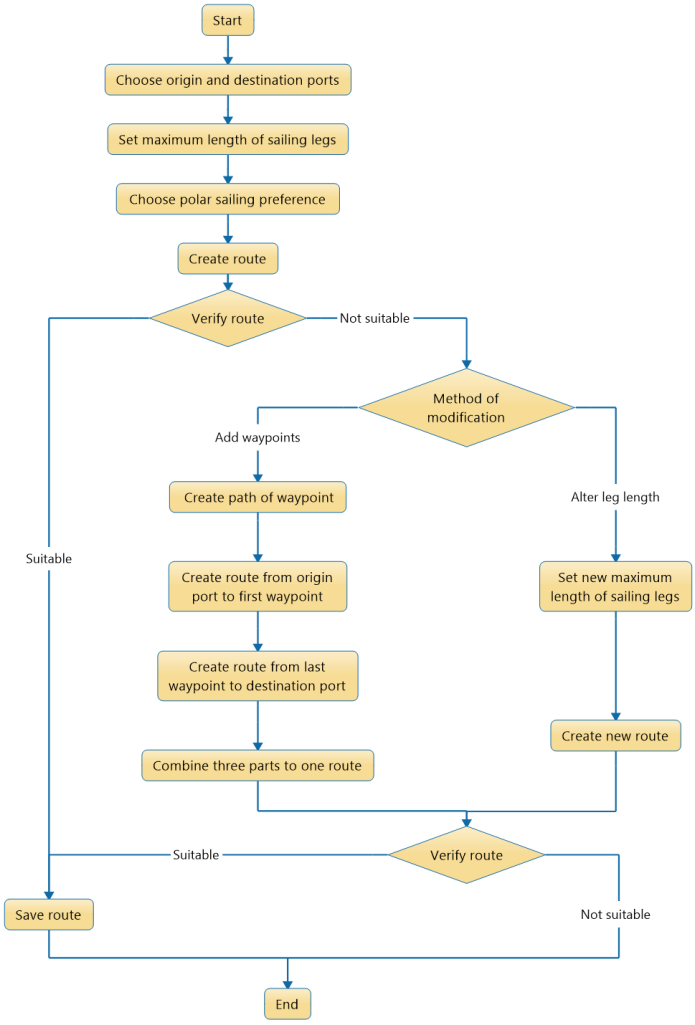


Figure 3.3: The overarching model as experienced by the user

3.2 Model overview

The route-finding algorithm is executed by running a MATLAB script named *RouteCreator*. Within the script is an overarching algorithm whose purpose is to read user defined input, run the route-finding algorithm, and then let the user verify the resulting route. An option to modify the route is also included. A flow chart of the overarching algorithm can be seen in figure 3.3.

When the script is executed, the user is fist prompted to define the origin and destination ports. This can be done by choosing one of the ports included in the prompted list or by manual input. The geographical coordinates as well as a name for each port, is stored as variables in the MATLAB workspace. Next, the user is asked to set a desired maximum length of the sailing legs, and a preference for either allowing or restricting routes going through polar waters. The function containing the route-finding algorithm then generates a route between ports omitting all land areas based on the given input. A figure showing the resulting route on a map is created, and the user is asked to visually inspect and verify the route. Three options are then given. If the route is suitable for use it can be saved for later use as input in the simulation model. If the resulting route is not suitable for use, the user can choose one of two methods to modify the route.

The fist method of modification is to manually add a single waypoint or a path of waypoint, which the new route is forced to include. If this option is selected, the map of the route is displayed, and the user is prompted to click on desired locations for the new waypoints. Then two routes are created, one from the origin port to the first waypoint, and one from the last waypoint to the destination port. The two routes and the waypoints are then merged to form the new proposed route. The other alternative is to change the maximum length of the sailing legs. If this option is chosen, a new maximum length must be set by the user. A new route is then generated by running the route-finding algorithm again with the new input value.

For either method the modified route is displayed on a map. At this point the route can be saved or discarded. If the latter is chosen, no route is saved, and the script is terminated.

3.2.1 User defined input

The process of defining model input is illustrated in figure 3.4. In order to set up a route, the coordinates of the desired origin and destination ports are needed. A list containing a selection of 20 large ports have been generated. The user is first prompted to select origin port and thereafter the destination port. Alternatively, a custom port can be defined manually by clicking the button ”manual input”. The latter option requires latitude and longitude coordinates, together with a name for the port. Either method can be used for either port.

When selecting a port from the included list, a pre-defined path of waypoints from the port to open waters is also loaded in most cases. The route-finding algorithm will then generate a route between the end points of each path. Defining a path towards open waters is not an option when choosing a custom port. It is therefore important that the latitude and longitude coordinates used for custom ports, describe a location with access to the open seas. This means that it must be possible to sail from the given location to the open sea along a rhumb line without intersecting land. Furthermore, the set of possible rhumb line

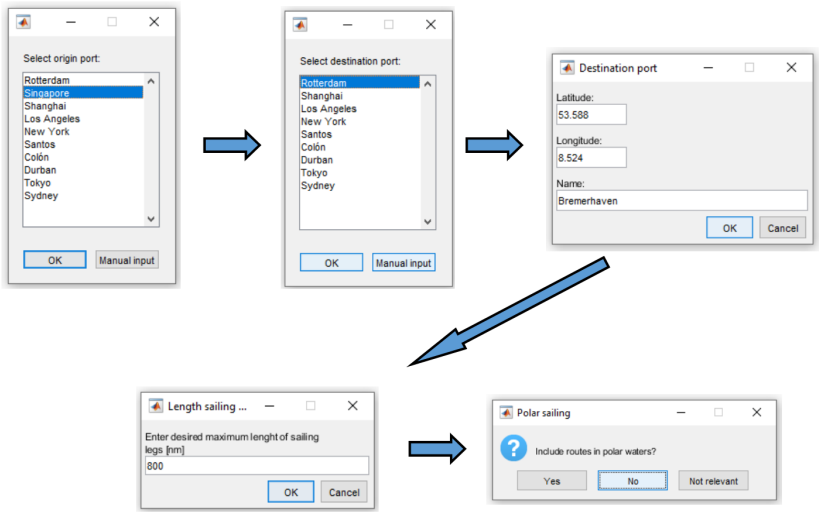


Figure 3.4: User defined input

paths towards the open sea should span an area as large as possible. The model might not work correctly if the port locations are set slightly inland. Due to time-constraints great focus has not been devoted to creating a large database of ports. The included ports are selected so that all continents are represented, with the intention to test the route-finding algorithm with a diverse set of routes.

The dialogue boxes used to set a desired maximum length of the sailing legs, and then select an option for restricting routes in polar environments, are seen in the lower part of figure 3.4. The maximum length of the distance between waypoints, will determine the discretisation of the routes between two land areas and between ports and land areas. The value should be given in nautical miles. Distances between waypoints along the shorelines will not be affected, since these are described by sailing paths predefined by the input data. The option for whether to consider routes in polar waters, is set by clicking either ”yes” or ”no” in the dialogue box. If such limitation is chosen, the routes will not go beyond latitudes of 70 degrees north and 60 degrees south. In all cases the routes are restricted from going beyond 82 degrees north.

3.3 The basic route-finding algorithm: creating a route around one land area

The algorithm for creating a route while circumventing a single land area is described in the following. First an overview is presented, then details of each step of the algorithm is highlighted. Figure 3.6 illustrates the process of finding a route circumventing land when the point of origin and destination lie on opposite sides of the land area. A flow chart of the algorithm is shown figure 3.5.

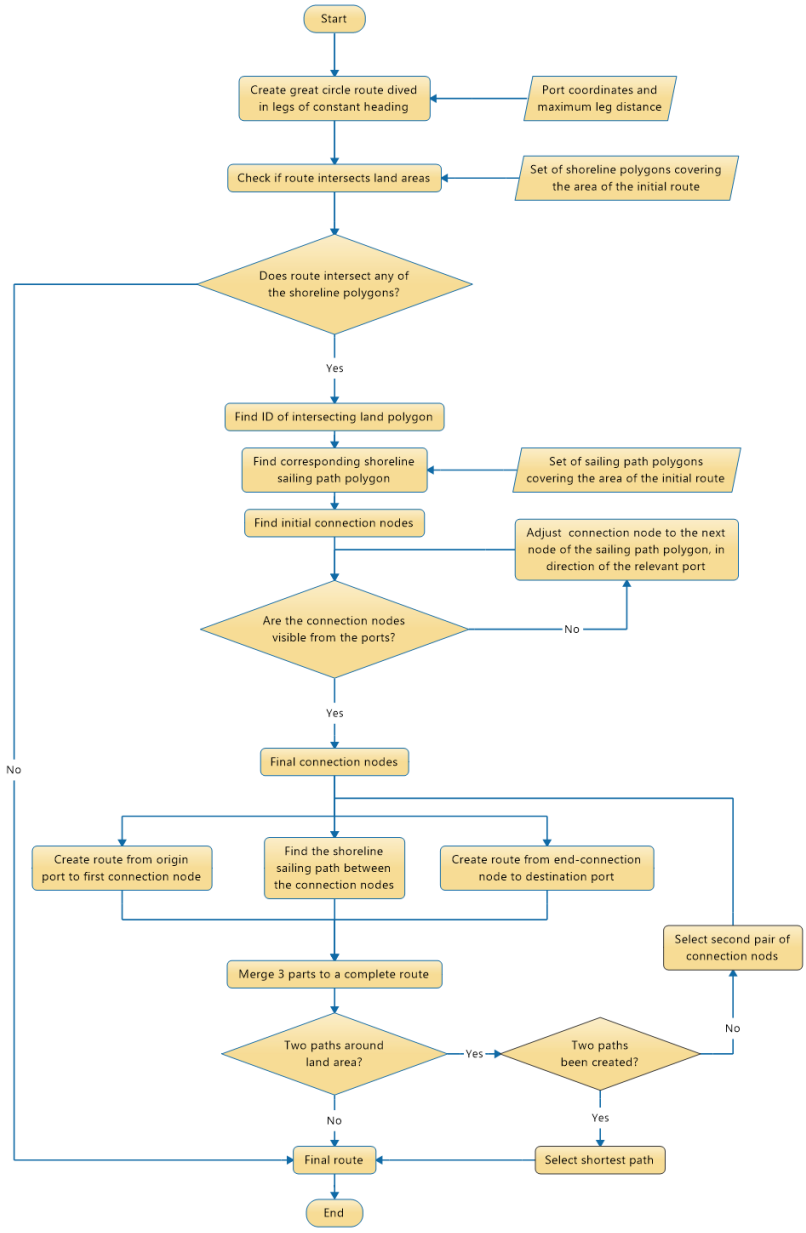
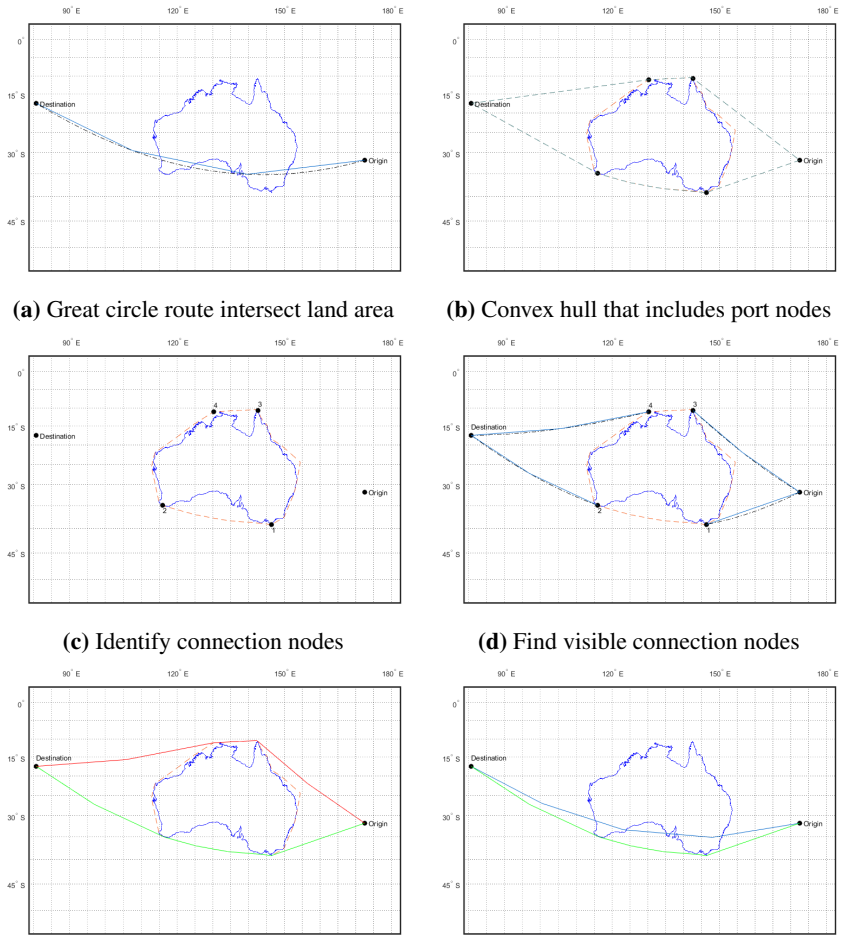


Figure 3.5: Flow chart of the route-finding algorithm for circumventing one land area.

Initially the algorithm sets up a great circle route, divided in a given number n of waypoints, resulting in $n-1$ sailing legs of constant heading. The number of legs is calculated according to the desired maximum length of each leg. The algorithm then

checks if the resulting route intersects with any of the land areas described by the shoreline polygons. If an intersection is detected, a specific ID for the land area to be omitted returned. When the desired shoreline polygon is identified, the corresponding shoreline sailing path polygon is retrieved. The next step locates the initial connection nodes. These are nodes of the shoreline sailing path polygon suitable for connecting the paths from each port, to the shoreline sailing path.



(a) Great circle route intersect land area

(b) Convex hull that includes port nodes

(c) Identify connection nodes

(d) Find visible connection nodes

(e) Two paths around land area

(f) Identify shortest path

Figure 3.6: Basic model for route generation, when great circle route intersects one land area

The algorithm will create either one or two routes around the land area, depending on the location of the ports relative to each other and to the land area. All routes are made up of three parts. One part from the origin port to the coast of the land area, one part along the coast of the land area, and a final part from the land area to the destination point. The paths between the ports and the respective connection nodes are created as rhumb lines

between the waypoints of a great circle route. While the path along the land area is given from the related shoreline sailing path polygon.

With the initial connection nodes established, the algorithm next ensures that the connection nodes are visible from each port. In this context, visible means that the great circle route from the connection nodes to the ports does not intersect the current shoreline polygon. If the connection nodes are not visible, they are adjusted until they are. When the visibility has been ensured, one or two paths are created by merging the three parts of the route. If two paths are created, the shortest path is selected as the desired route.

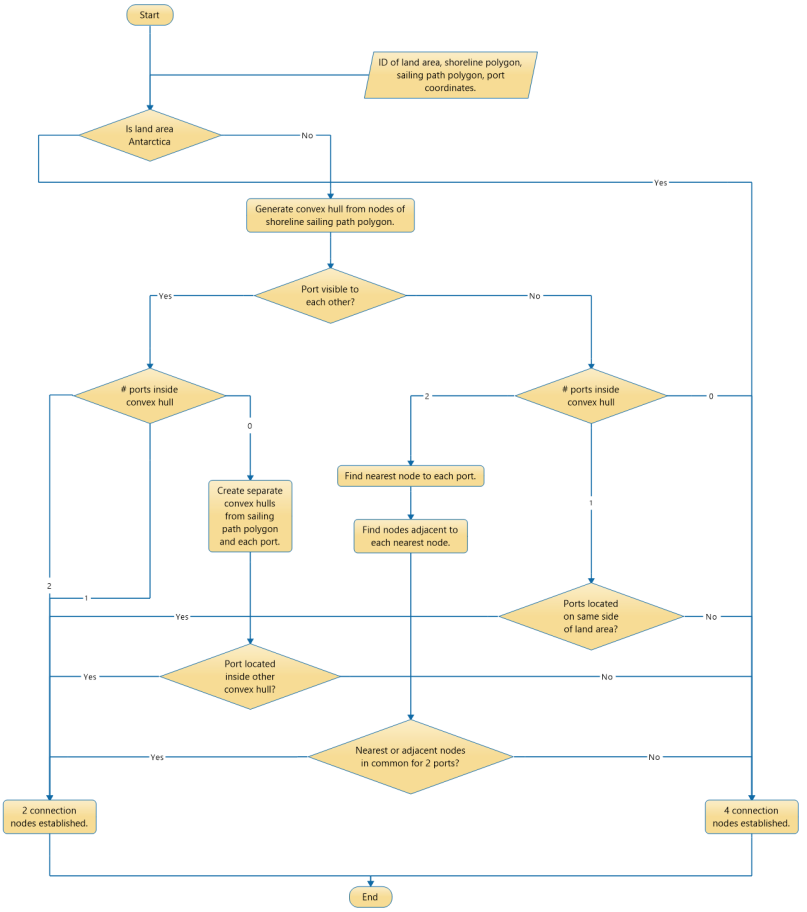


Figure 3.7: Number of connection nodes established in different circumstances

3.3.1 Finding initial connection nodes

When a route intersects a land area, one or several connection nodes must be identified in order to connect the open seas part of the route to the shoreline sailing path. The

number of paths generated is decided by the number of connection nodes established. Four connection nods are found if two routes are to be created, and two otherwise. The shoreline sailing path can be a single node of the sailing path polygon. In such cases the initial connection nodes for that path, will be equal for both ports.

The port locations relative to each other and to the land area, determines if it is sensible to create one or two paths. The port locations are evaluated by determining if any of the ports lie inside a convex hull around the given shoreline polygon, and if the ports are visible to each other by a rhumb line. Figure 3.7 illustrates the different conditions resulting in two or four connection nodes.

Two connection nodes are generally found when the ports are visible to each other. The two exceptions are when the land area in question is the Antarctic Continent, and when the ports are sufficiently distant from the land area. Four connection nodes are established in those cases. When the ports are not visible to each other, there are two instances resulting in two connection nodes. The first is when both ports are located inside the convex hull, and the ports share a nearest node, or nodes adjacent to the nearest node to each port. The other instance is when only one port is located inside the convex hull, but the ports still lie on the same side of the land area. Otherwise four connection nodes are established.

Combining port locations and port visibility results in six main conditions for determining the connection nodes, in addition to the separate event of circumventing the Antarctic continent. For each of the seven possibilities there are further considerations determining the approach. A comprehensive flow chart of the process of determining connection nodes and generating routes around land areas, is included in electronic appendix. The algorithms for finding connection nodes are coded in six different MATLAB functions, and is described in the following sections.

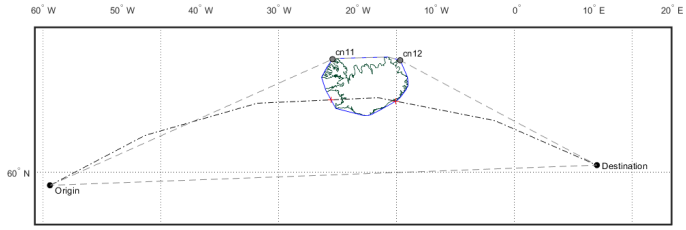
Both ports are outside of the convex hull and not visible to each other

The most basic case is when the ports lie on opposite sides of the land area, and both are located outside a convex hull around the land area. In this situation the connection nodes are found by generating a convex hull around the origin port, the sailing path polygon and the destination port, as demonstrated in figure 3.6b. The desired nodes are then simply the nodes adjacent to each port node. The connection nodes are given as the index of the same nodes in the sailing path polygon.

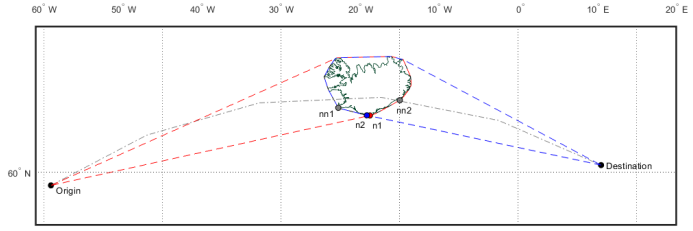
Both ports are outside of the convex hull and visible to each other

A slightly more complex situation is encountered when the ports are visible to each other by a rhumb line, while a great circle route between them intersects a land area. Figure 3.8a illustrates one such case. Following the same procedure in this case will not yield feasible connection nodes, as one of the adjacent nodes to each port node will be the other port node. Connection nodes would thus only be found for one route omitting the land area on the opposite side of the two ports. While a shorter path might be found on the same side of the land area where the ports lie. Sailing along a rhumb line between the ports is not a desired solution either, since the objective is to find the near shortest route between ports. The problem is solved by finding connection nodes for a second path, omitting the land

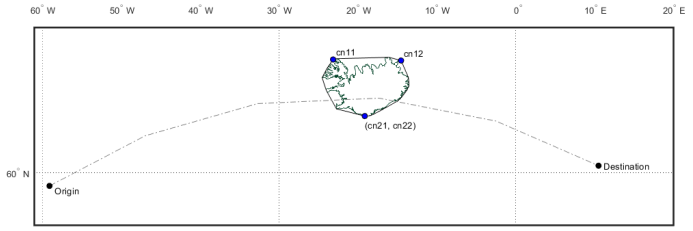
area on the side where the ports are located. The two connection nodes for the path along the opposite side of the land area, are kept as found from the initial convex hull.



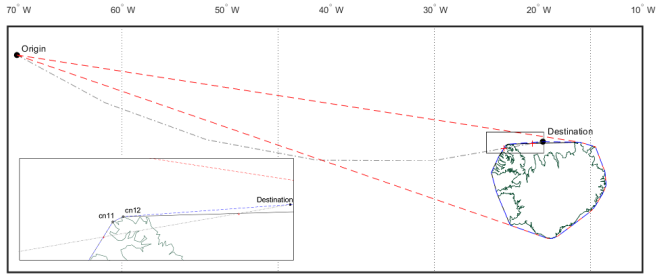
(a) Single convex hull yields connection nodes for one path only



(b) Potential connection nodes for second path from two convex hulls and points of intersection



(c) Resulting connection nodes. Path 2 has identical connection nodes for both ports



(d) Two connection nodes, from point of intersection, when one port lie near shore

Figure 3.8: Both ports lie outside sailing path polygon. Ports are visible to each other, while a great circle route between them intersects a land area

Three options for connection nodes are considered for the other path. First the points of intersection between the great circle route and the sailing path polygon is determined, indicated by the red crosses in figure 3.8a. Then the nearest node to each point of intersection is located as possible connection nodes. Two additional possible connection nodes are established by generating convex hulls around the sailing path polygon, and each port separately as illustrated in figure 3.8b. Each convex hull results in one candidate as connection node, identical for both ports. The connection nodes yielding the shortest path are selected. The resulting connection nodes are illustrated in figure 3.8c.

A special case can arise if one port is located near shore, as illustrated in figure 3.8d. Here one port is located within the convex hull created from the sailing path polygon and the other port. In such cases it is clearly longer to sail around the land area, and only the two connection nodes found from the points of intersection are used.

Both ports located inside the convex hull and are not visible to each other

When both ports are located inside the convex hull, the connection nodes are found from inspecting the node of the sailing path polygon nearest to each port, and the nodes adjacent to each nearest node. One adjacent node in each direction is inspected for polygons with less than ten nodes, while two adjacent nodes are included for polygons with ten or more nodes. For large polygons with more than 50 nodes, three adjacent nodes in each direction is investigated. If the ports share a common nearest node, or have any of the adjacent nodes in common, two connection nodes are found. Otherwise four connection nodes are established.

(a) **(b)** **(c)**

Figure 3.9: Connection nodes when both ports are located inside a convex hull, and the ports are not visible to each other

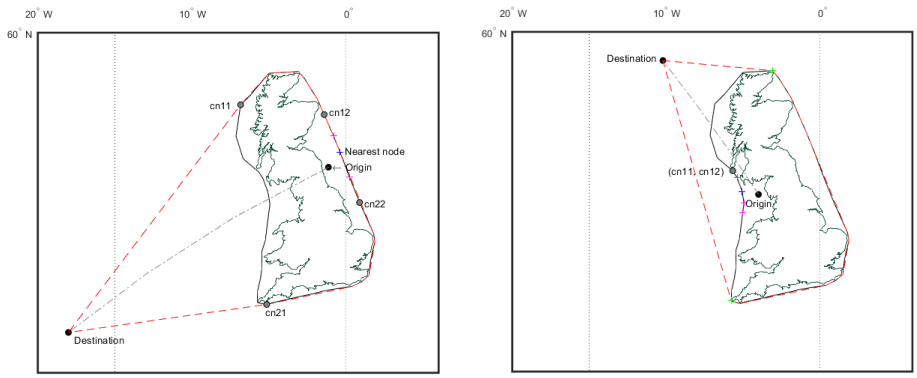
Figure 3.9a illustrates a case when the ports does not have a nearest node or any of its adjacent nodes in common. The furthest adjacent node in each direction, for each port, is then set as connection nodes.

If the nearest node is same node for both ports, that node is set as connection node for both ports. Similarly, if the ports share one of the adjacent nodes. In some cases, the ports can have several of the adjacent nodes in common. An example can be seen in figure 3.9b. Then for each common node, the distance of a route from the origin port to the given node and onward to the destination port is calculated. The node yielding the shortest distance route is set as connection node for both ports. The shared nearest nodes, and the chosen connection node, is displayed in figure 3.9c.

One ports lie inside the convex hull and are not visible to each other

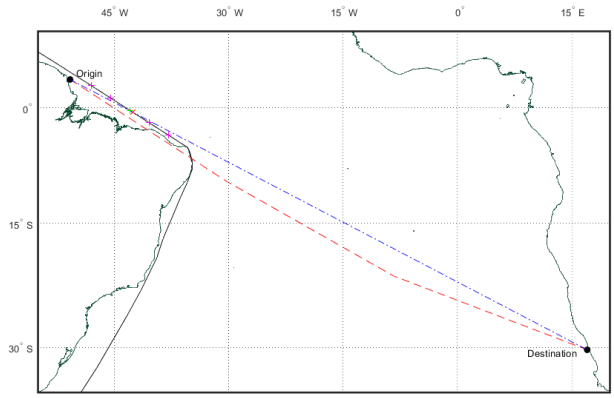
When one port is located inside and the other outside the convex hull, the connection nodes for the inside port is again found by finding the nodes adjacent to the sailing path node nearest the port in question. The connection nodes for the other port is found by generating a convex hull with the port and the sailing path polygon. The connection nodes are found as the nodes of the sailing path polygon equal to the two nodes adjacent to the port node in the convex hull. Figure 3.10a demonstrates this process.

A challenge occurs when the ports lie on the same side of a land polygon but are not visible to each other, as illustrated in figure 3.10b. When this occurs, a rhumb line route between the ports typically intersects a small part of the land area. One path is suitable in such cases and two identical connection nodes are returned. This situation is checked for by inspecting the indices of the connection nodes. If the ports outside and inside the convex hull share a common node, that node is chosen as connection node for both ports. And when the connection nodes for the port outside the convex hull lie beyond the connection nodes for the inside port, the nearest or adjacent node yielding the shortest distance path is chosen as connection node.

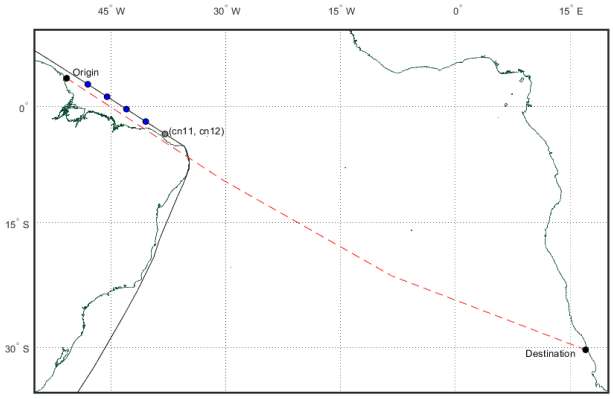


(a) Ports located on opposite sides a land area. **(b)** Ports located on the same side of a land area.

Figure 3.10: Connection nodes when one port lie inside the convex hull, and the ports are not visible to each other



(a) Nearest node to point of intersection, and adjacent nodes in each direction.



(b) Resulting connection node among the possible candidate nodes.

Figure 3.11: Connection nodes when one port is located inside the sailing path polygon, and the ports are visible to each other by a rhumb line.

One or both ports lie inside convex hull, and are visible to each other

In some cases, the ports can be visible to each other by a rhumb line, while one or both ports lie inside a convex hull around a land area. The algorithm accounts for five such possibilities, which are briefly described below. In all cases a path around the land area will clearly not be the shortest path, and only two connection nodes are thus established. In some of the instances a rhumb line route could be a better option than a route via the shoreline sailing path polygon, but connection nodes are still found in order to comply with the overall algorithm.

A special case can arise when both ports are located on the border or the sailing path polygon. In such cases the connection nodes are set as the node of the sailing path polygon nearest each port, but in the direction of the other port.

Another possibility is that both ports either lie inside the sailing path polygon, or that

both lie inside the convex hull while outside the sailing path polygon. The connection nodes are then established by inspecting the nearest node to each port and its adjacent nodes. The process is like that previously described for two ports located inside the convex hull while not being visible to each other. If the ports do not share the same nearest node or any adjacent nodes, two different nodes are assigned. Otherwise the connection nodes are equal for both ports.

When one port is located on the border of the shoreline sailing path polygon with the other port outside, the two nodes nearest the former port are identified. And the node yielding the shortest distance path is set as the connection node for both ports.

A fourth possible scenario is when only one port is located inside the sailing path polygon with the other port located outside. The latter can be either inside or outside the convex hull around the land area. In such instances, the node nearest the point of intersection between the sailing path polygon and a rhumb line between the ports is found. In figure 3.11a, the point is marked by the red ”x”, and the nearest node is shown as a blue cross. The connection node is then found by searching the nearest node and its adjacent nodes for the shortest total distance. Figure 3.11b illustrated how the connection node resulting in the shortest distance route, is selected as connection node for both ports.

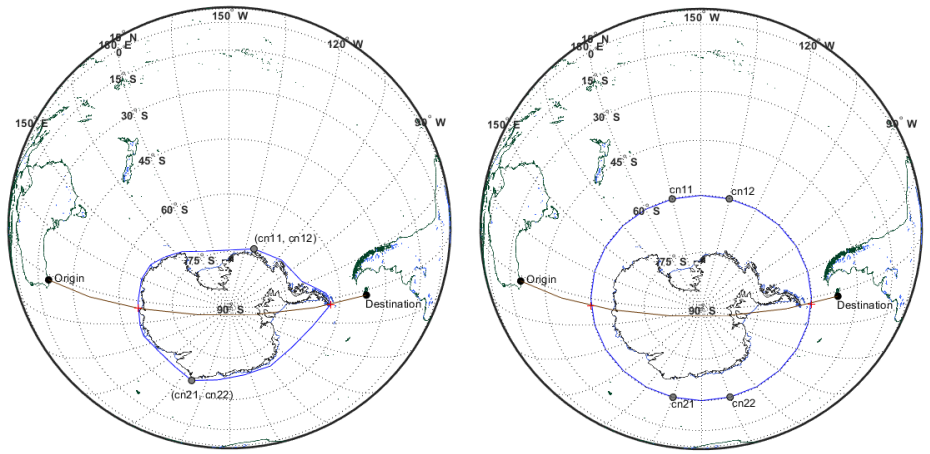
A final special case is when one port is located inside the convex hull while being outside the sailing path polygon, and the other port is located outside the convex hull. In such instances the algorithm first identifies the point of intersection between the sailing path polygon and the initial route between the two ports. The connection nodes are then set as the nodes of the sailing path polygon nearest each point of intersection.

Connection nodes for Antarctica

A special case arises when the route is crossing the Antarctic Continent. The two ports will be visible to each other in most cases, and in any case where both ports are located outside a convex hull around the land polygon. The ports can lie on opposite sides of the continent on a globe, but still be visible to each other by a rhumb line. Generating a convex hull with the ports as nodes would not be a fruitful approach, since the continent is spanning the complete longitude range. Generating two convex hulls would result in connection nodes around the $-180°$ and the $180°$ meridian, for all sets of ports located outside the continent.

This conundrum is solved by finding two identical connection nodes for both ports, and for two paths around the continent. First the two nodes of the sailing path polygon nearest the two points of intersection between the shoreline polygon and the great circle route is identified. Thereafter the middle node in between the two nodes, is located. This is done in both directions such that two middle nodes, on in each direction, are then set as the connection nodes for each path. The process is illustrated in figure 3.12a.

If polar sailing is restricted while the route intersects the Antarctic Continent, the sailing path polygon around Antarctica is altered to follow a path of constant latitude of 60 degrees south. The connection nodes are also modified in such cases, to minimise the risk of great circle routes from the connection node to the ports going below the set latitude limit. The two connection nodes adjacent the previously found middle nodes, as set as the connection nodes for each path. Figure 3.12b shows how four unique connection nodes are found when sailing in polar water is restricted.



(a) Polar sailing allowed **(b)** Polar sailing restricted

Figure 3.12: Connection nodes for circumventing the Antarctic Continent.

3.3.2 Visibility test of the connection nodes

All routes in open water are generated as great circle routes divided in waypoints, resulting in an approximate great circle route divided in legs of constant heading. It is possible that the ports and corresponding initial connection nodes are visible to each other by a rhumb line, but not visible by the approximate great circle route. An example of this is illustrated in 3.13a. The initial connection nodes are therefore subjected to a visibility test, to see if the route between a port and the connection nodes intersects the land area to be omitted. When an intersection is detected the given connection node is moved to the adjacent node towards the related port. The process is continued until the connection node is visible to the relevant port, as illustrated in figure 3.13b. The resulting nodes are set as final connection nodes, when all nodes have been tested, and adjusted if needed.

When two paths are created, path one will always have a clockwise direction, while the opposite is true for path two. The node index of path one will therefore increase along the direction of travel, while the opposite is true for path two. If only on path is created, the direction of travel along the sailing path polygon must be identified prior to the visibility check, to ensure correct results.

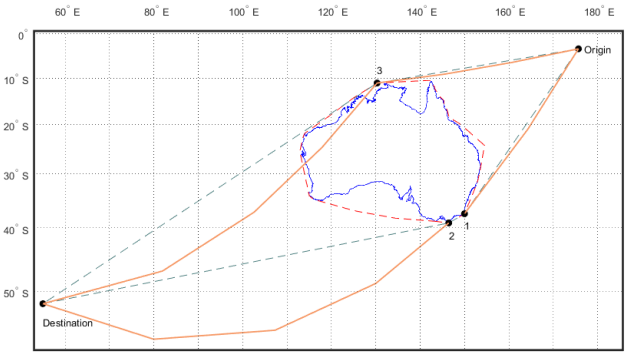
3.3.3 Constructing paths around land areas

After the final connection nodes are determined, one or two paths around the land area will be created. The basic method for constructing the paths is equal in both cases. The route is made up of three parts: a path from the origin port to the coast of the intersected land area, a path along the sailing path polygon surrounding the land area, and a final part from the coast of land area to the destination port. The task to be performed is thus to extract the path along the land area in a correct manner in respect to the indices of the nodes of the

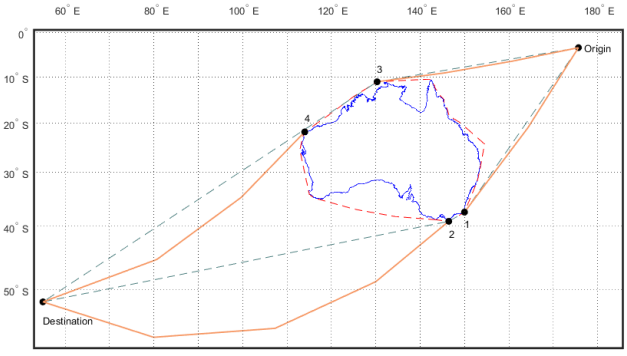
sailing path polygon and the direction of the route. Thereafter it is quite simple to combine the three parts to a complete route.

Sometimes a port can be a node in the sailing path polygon around the intersected land area. In such cases, the connection node is set as the port node. Also, a path from the port to the connection node is not necessary and is thus set to be void.

The model includes an option for restricting routes in polar waters. For routes crossing the Antarctic continent the restrictions are implemented in the process of finding the connection nodes. Since there is no Arctic continent, a separate solution has been implemented for routes involving the Arctic. A route in the Arctic will only be generated as one of two paths around a land area, unless one or both ports are located in the Arctic. When polar sailing is restricted, the algorithm for generating two paths multiplies the distance of any route going further north than 60 degrees latitude by a factor of 100. This is a simple way to ensure that the other path is selected as the shorter path.



(a) Initial connection nodes from convex hull



(b) Final connection nodes after visibility check

Figure 3.13: Initial connection nodes are subjected to a visibility test

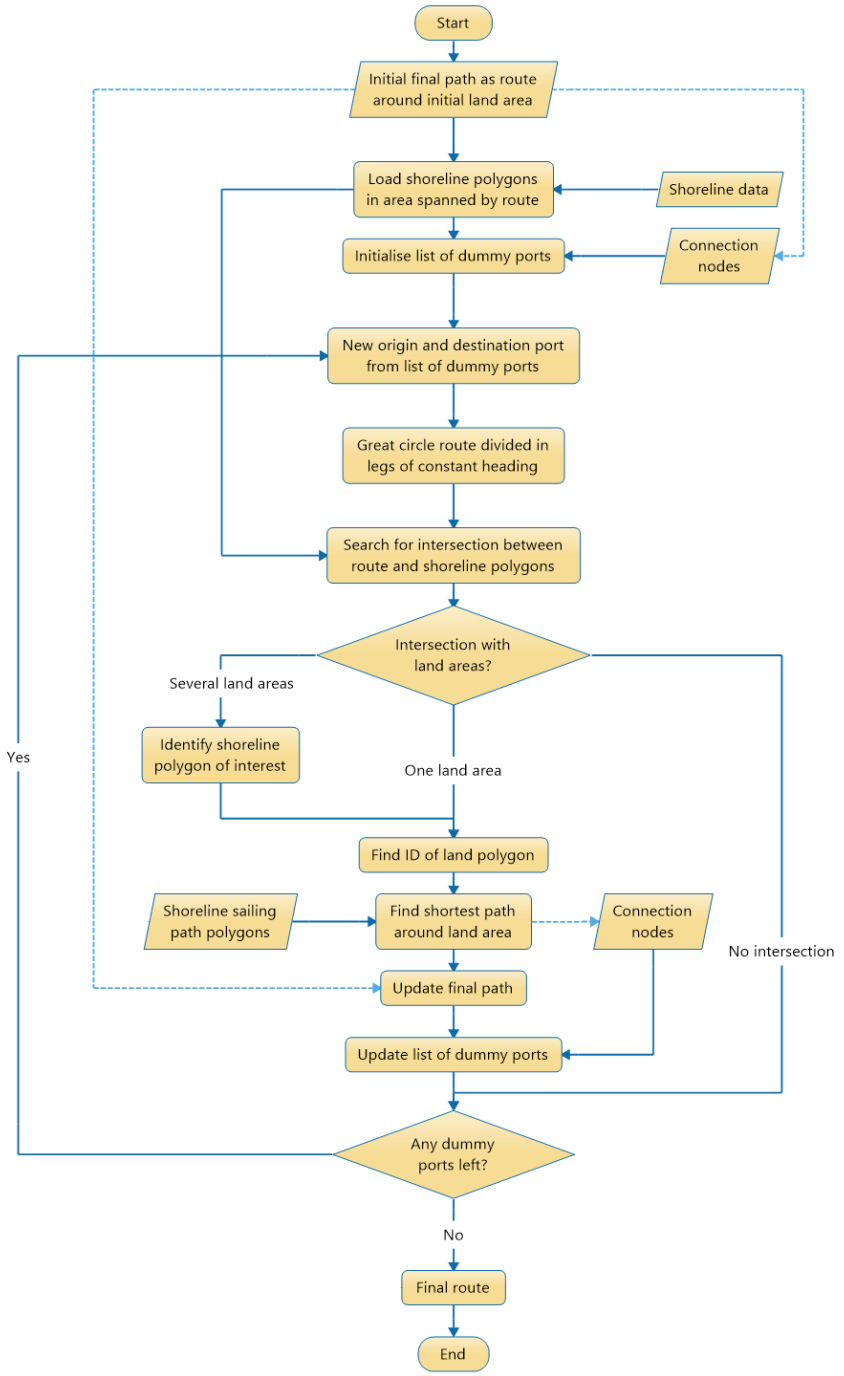


Figure 3.14: Flow chart of the complete route-finding algorithm, starting with an initial route circumventing one land area

3.4 The complete route-finding algorithm

The complete route-finding algorithm creates a route between two ports, circumventing all land areas. This is done by an iterative process of checking if an already established route around one land area intersects other land areas, and if so alter one part of the route at the time, until there is no more intersections between the final route and the shoreline polygons. The complete algorithm is an expansion of the algorithm for finding a route circumventing one land area described in the previous section.

The algorithm starts by establishing an approximate great circle route between the ports, divided in waypoints and rhumb lines. The route is described by the waypoint coordinates and the course and distance of each rhumb line. Shoreline data is loaded for an area covering the initial route, including a $10°$ buffer in each direction. If a shoreline polygon has any nodes with coordinates within the defined area, the complete polygon is included in the shoreline data.

The initial route is checked against all the included shoreline polygons for intersections. In the case of several intersection land areas, a choice is made of which land area to first circumvent. The chosen land area is identified by a specific ID for the relevant shoreline polygon. A route is created between the ports omitting the selected land area in the manner described in section 3.3. The resulting route is stored as a candidate for the final path.

A flow chart of the extended algorithm, starting with the initial route omitting one land area, is displayed in figure 3.14. After the first candidate route has been established, the shoreline data is loaded anew to potentially cover a larger area than was spanned by the initial great circle route. Then a list of dummy ports is created, and the iterative process begins.

The dummy ports are nodes of the previously found candidate route. These nodes represent endpoints for the parts of the route crossing open waters and are used as points of origin and destination in the basic route-finding algorithm. The list of dummy port contains latitude and longitude coordinates for two nodes per entry and is initialised with two entries. These represent the coordinates of the origin port and the first connection node, and the coordinates of the second connection node and the destination port. The dummy port list is updated when parts of the route are modified to omit additional intersecting land areas.

Next, the algorithm selects a set of dummy ports from the list. Starting with the first entry, the first dummy ports are the origin port and its adjacent connection node, from the initial route. The basic route-finding algorithm is then repeated for the current set of dummy port. Thus, a great circle route between the dummy ports is established, and the route is checked for intersections with any shoreline polygons. If one or several polygons are intersected, the ID of a specific chosen shoreline polygon is retrieved. A new route between the dummy ports, circumventing the identified land area is found. And the relevant part of the initial route, the distance between the two dummy ports, is updated. At this point there are two new connection nodes in the updated route and the list of dummy ports is updated to include those. Then a new iteration starts by choosing a new set of dummy ports from the updated list.

If the established route between the current dummy ports does not intersect any land area, the route is kept as is. The process will advance to a new iteration as long as there

are more dummy ports in the list. If so, a new set of dummy ports are chosen from the list of dummy ports, and the process repeats until the routes between all sets of dummy ports have been inspected. At this point a final route circumventing all areas have been established. The route is described by the coordinates of all waypoints, and the distance and course of each leg.

3.4.1 Selection of shoreline polygon

There are several considerations that concerns the choice of shoreline polygon to first be circumvented. The algorithm is intended to select land areas in a manner that avoids generating a longer route than necessary, by including a route segment around a land area that would not be intersected by a route around the lager area. To this intent, the algorithm checking for intersections between the route and the shoreline polygons specifically register if an intersected polygon is a continent. If none of the intersected polygons are continents, the polygon representing the largest land area is selected. Should one of the intersected polygons be a continent, that polygon is selected.

In the case when several of the intersected areas are continents, the algorithm checks if the continents also are intersected by a rhumb line between the ports. This is done to avoid generating routes around a continent only intersected by a great circle route when the ports are located on opposite sides of another continent. When none of the intersected continents are also intersected by a rhumb line, the polygon representing the largest area continent is chosen. If one of the intersected continents are also intersected by a rhumb line route, that polygon is selected.

A special consideration arises when several continents are intersected by a rhumb line route between the ports. In such cases the rhumb line distance spanning each continent is calculated, and two polygons with the greatest span are inspected. If the polygon with the largest span also represent the largest area, that polygon is selected. Also, if rhumb line distance spanning one polygon is twice or more the distance spanning the other polygon, the former is chosen. Otherwise the largest area polygon is selected.

Chapter 4

Simulation theory and model background

This chapter presents a brief introduction to simulation theory, followed by some background information relevant to the simulation model.

4.1 Simulation theory basics

The motivation for applying simulation is usually a desire to study a system and to gather information about how a specific system works. A system can be defined as a collection of entities that interact with each other to accomplish a logical end(Law, 2007). Examples of systems can be anything from a production line in a factory to the luggage handling at an airport, or a fleet of vessels transporting goods.

Besides studying a system by gathering information from the system running in its normal mode of operation, additional knowledge can be gained by studying how a system responds to alternative configurations. In other words, by experimenting with the system configuration. Experimenting with a system itself is a certain way to gather information valid for the specific system. Practical or ethical restriction, or cost and time demand often make this option unfeasible. An alternative in such instances, is to make a model of the system and to rather study the model. The model can be either a physical model or a mathematical model. A mathematical model can be solved analytically if it is not too complex. Gosavi (2015) defines a simulation model as a computer program mimicking a systems behaviour. And further states that simulation models can be used to model large, complex stochastic systems, for which constructing mathematical model can be difficult.

A simulation model will always be an ideal representation of the real system and will therefore never be quite exact. A good simulation model can nevertheless reveal useful information about how a system works, as well as predictions to how a system will respond to altered conditions.

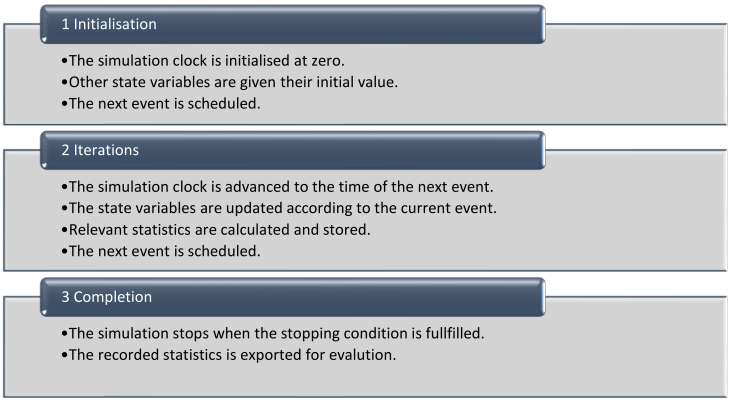


Figure 4.1: General steps of a discrete-event simulation

4.1.1 Discrete-event simulation

There are several possible ways to make a simulation model. In a discrete-event simulation model, changes to the state of the system is represented by a collection of discrete events(Fishman, 2001). Law (2007) defines the state of system as the collection of state variables necessary to describe the system. A discrete-event simulation is thus a simulation, in which the state variables changes values instantaneously at separate points in time. An obvious advantage of using discrete-event simulation is that the simulation skips the time between events, greatly reducing the time needed to run a simulation, as compared to the real-life process.

Each change in state variables is called an event(Fishman, 2001). The simulation clock is a state variable which is equal to the current value of the simulated time. The simulation clock jumps in time from the time of one event to the time of the following event, and the system state remains constant between events. At each event the state of the system is updated corresponding to the given event. Using an event-scheduling approach involves planning the next event at each current event. When the state variables have been updated, the simulation clock is advanced to the time of the next event, and that event is executed. The system state is again updated, and the next event is scheduled. A simulation keeps advancing from one event to the next, until a stop condition is met. The stop condition can be a time limit, a specific criterion for some simulation variable, or the occurrence of some special event. The general steps of a discrete-event simulation are summarised in figure 4.1.

4.1.2 Verification and validation of simulation models

The following is based on an article by Sargent (2005), wherein he presents and discuss the topic of verification and validation of simulation models. According to the article, verification is related to the correct functioning and implementation of the computer model, while validation of the model is related to the accuracy of the output of the simulation

model. To validate a model is thus to ensure that the result have an acceptable accuracy, and that the model therefore gives valuable information in accordance with its indented purpose. The required accuracy of the output variables will normally have to be specified in order to validate the model. According to Sargent (2005) this should be done at an early stage of development.

Sargent (2005) presents a simplified version of the modelling process, where a conceptual model of the system is first created, before a computer model can be built based on the conceptual model. Four types of validation and verification is discussed. These should be performed at each step of the modelling process.

Conceptual model validation involves validating that the conceptual model is a reasonable representation of the system, for the intended purpose of the model. This includes testing the underlying theories and assumptions used to create the model, to ensure they are appropriate. Mathematical analysis and statistical methods should be used on the system data utilised to create the model. Processes are often described by stochastic variables and statistical distributions. In conceptual model validation, such statistical assumptions regarding the characteristics of the system processes, should be validated by analysing the system data. A crucial part of conceptual model validation is to ensure that the model's structure and logical relationships echoes those of the system, or is a reasonable representation for the purpose of the model.

"*Computerised model verification* is defined as assuring that the computer programming and implementation of the conceptual model is correct."(Sargent, 2005). Techniques for this purpose include structured walk-throughs of the computer model, tracing entities through a model to ensure correct model logic, inspecting input-output relationships and internal consistency checks.

A large part of the evaluation of a simulation model takes place as *operational validation*. Sargent (2005) defines operational validation as ensuring that the model's output behaviour has sufficient accuracy for the model's intended purpose. To that end the models output behaviour should be compared to that of the system. Preferably for several different sets of experimental conditions, in order to obtain a high degree of confidence in the model and its results. It is therefore not possible to achieve high confidence if it is not possible to collect operational behaviour system data. In such instances Sargent (2005) recommends a thorough evaluation of the models output behaviour, and if possible, comparison to other valid simulation models.

System data is needed in all stages of modelling and experimenting. Therefore *data validity* is essential for validation and verification throughout the modelling process. Sargent (2005) defines data validity as "ensuring that the data necessary for the model building, model evaluation and testing, and conducting the model experiments to solve the problem are adequate and correct". It is not possible to obtain high model confidence, as a rule, without sufficient valid system data.

4.2 Simulation model background

Some background information relevant to the simulation model is presented in the following. First a short presentation of the original simulation model is given. Then follows a description of the data used to model weather conditions or sea states in the simulation

model, and a description of the software used.

4.2.1 The original simulation model

A detailed description of the original simulation model is presented in the thesis of Bakke and Tenfjord (2017) and will not be restated here. Rather some relevant aspects of the model are discussed in the following paragraphs.

The original simulation model requires a fixed route, with weather data covering each leg of the route, organised in separate spread sheets. The simulation model thus requires some degree of cumbersome pre-processing. In order to set up a new simulation in the existing model, the user must create a route divided in legs, and then download weather information covering each area and store it as input in the correct format. The simulation model could be significantly more flexible if the required weather data could be stored in a single file. As an example, global historic weather data spanning several years could be included as input. Then the relevant data for an area covering the current route, and for the time period to be simulated could be extracted prior to the simulation.

In the simulation model, the weather conditions are modelled in a stochastic manner. A method that requires creating transition probability matrices to determine the next state of the system based on the current state. When running a simulation, transition probability matrices for each leg of the journey are created. These are based on weather data of an area covering each leg, for a certain time period.

There are several arguments for why such method fails to model the weather conditions in a realistic manner. Local variations in weather conditions will be poorly reflected, since the transition probabilities are equal for a rather large area, and since they do not vary over time. Using Markov chains can also result in abrupt changes in the different variables describing the wind and wave conditions. While such conditions can change rapidly in nature, purely probabilistic determination does not reflect natural variations over time in a good way. Another issue is that the different variables can vary independently in a not so realistic manner.

Finally, with the Markov chain approach, the weather variables can only take a discrete set of values. This can result in some of the simulated performance parameters varying in a discrete manner, which would not be reflective of real-life behaviour. For the above-mentioned reasons, it seems like better approach to apply historic weather data directly in the simulation model.

4.2.2 Met-ocean data

Met-ocean is a term used to described the combined effect of meteorology and oceanography(Chakrabarti, 2005). *Met-ocean data* is thus data describing the meteorological and oceanographic conditions. The simulation model applies hindcast weather data to emulate realistic conditions. Hindcast weather data is produced by a reanalysis of numerical weather prediction models constrained by observational datasets. The result of such reanalysis is weather data in gridded meteorological fields, consistent over the span of decades(Black and Henson, 2014).

The data used for creating the simulation model is obtained from the European Centre for Medium-Range Weather Forecasts (ECMWF), and is of Network Common Data

Form(*netCDF*) format. More details regarding the data is found on the ECMWF website(ECMWF, 2019).

4.2.3 Software

MATLAB and *Simulink* is the software used for creating the simulation model. While itself being created in Simulink, the simulation model is executed by running a MATLAB script. This allows for configuration of the simulation and loading of the prerequisite data into variables and parameters used by the simulation model.

Simulink and SimEvents

Simulink is a software environment, integrated with MATLAB, for creating models and performing simulations. The software provides a graphical interface for creating simulation models. The modelling is done by selecting from a library of blocks with different properties and functions. The blocks can be configured, and the relationship between different blocks can be set by connecting them in a desired order(MathWorks, 2019e).

SimEvents is a discrete-event simulation engine within Simulink, with a library of blocks for creating models of event-driven systems. Common blocks are sources and sinks for entities, queues, servers and switches. The entities can haven custom data attributes, that can be modified by blocks such as *entity servers*, according to the block configuration(MathWorks, 2019d).

An important feature of Simulink is the possibility to run MATLAB algorithms by applying *MATLAB function blocks*. The *Simulink function block* is another block of great utility. This block, which can be called from an entity server, can process input and return the resulting output. The two blocks can be combined to execute MATLAB algorithms with input variables given from an entity server, and can thus be used to perform complex operations on entity attributes.

NC-toolbox

Nctoolbox is MATLAB toolbox for reading common data model datasets. It can access files of several file formats, including netCDF files. Met-ocean data files can be quite large if the data covers a large area, a large time span or if the file has high resolution. Nctoolbox makes it possible to extract a subset of a variable from the met-ocean data file without loading the complete variable into MATLAB, thus reducing the time it takes to load the met-ocean data.

Chapter 5

Simulation model

In this chapter the simulation model is described. The idea behind the model is to simulate the sailing of a vessel along a fixed route, using discrete event simulation. Each event of the simulation represents a unique point along the route, with unique weather conditions for the current time and position. The first event represents the vessel at the origin port at the given start date. The simulation is performed by calculating the distance and time used to reach the next event, in the weather condition present at the current event. This is done by estimating the speed reduction resulting from the given wave conditions. The simulated journey is then advanced from point to point along the route until the destination is reached. At each event several vessel performance parameters are calculated based on the attainable speed and the current weather conditions. These are stored to be used for evaluating the performance of the simulated vessel.

5.1 Overall algorithm

The overall algorithm for running a simulation is coded in a MATLAB script named *run_sim*, included in appendix E.1.1. A flow chart of the process is shown in figure 5.1.

When script is executed, the first sub-process reads the user defined input. Thereafter a choice is given to either load a predefined route stored in a MATLAB file, or to generate a new route. When the latter is chosen a new route will be generated as to specifications. The user will then be prompted to inspect and verify the fitness of the resulting route, and if to proceed with the simulation. If the new route is deemed acceptable or when a predefined route is chosen, met-ocean data covering the given route is loaded as the next step of the process. Loading the met-ocean data can be resource demanding task, given the large amount of data needed to cover the given area of a route for a specific time period. Therefore, this data is only loaded if a simulation is to proceed.

With the relevant met-ocean data loaded, all prerequisite preparations for running the simulation is performed, and the next step is to run the Simulink model to perform the simulation. The Simulink model is described in section 5.2. When the simulation is done,

output data is processed and stored, before some output is plotted, and a map of the route is displayed. Finally, the user is given an option to save the results in a MATLAB file.

In the instances where a new route is created, and the route is deemed not suitable by the user, a warning will be displayed before the process will terminate without running a simulation. In such cases the user is advised to create a desirable route by running the RouteCreator script, and thereafter execute a new simulation with the resulting route given as input.

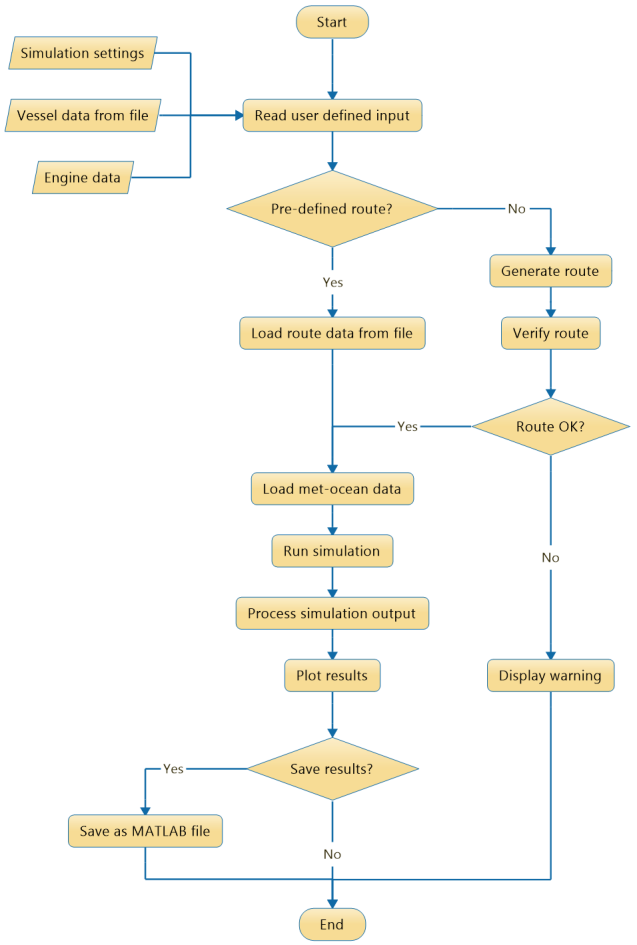


Figure 5.1: Algorithm for running the simulation model

5.1.1 Input

The model requires four types of input data. These are: manual input to configure the simulation, data describing the vessel whose performance is to be simulated, data describing

the ship engines, and met-ocean data. The latter is detailed in section 5.1.3, while other types of input is described in the following.

Manual input

The simulation model is configured by editing preferences in the MATLAB script for running a simulation. The configuration variables are listed in table 5.1. The start date of the simulation is given as year, month, day, hour, minute and second, and must be set within the time range covered by the given met-ocean data. The desired time step between simulation updates will in most cases equal the time between events in the simulation. This value should be given in hours.

Table 5.1: Configuration variables

Variable name	Description	Format
start_date	Starting date of the simulated journey	[yr,month,day,h,m,s]
time_step	Desired time step between simulation updates	[h]
speed_sailing	Desired sailing speed	[kts]
file_name_vessel	Name of file containing the vessel data	[Text string]
file_wind	Name of file containing wind data	[Text string]
file_wave	Name of file containing wave data	[Text string]

Vessel data

The vessel data is loaded to the MATLAB workspace from a spread sheet containing 54 parameters. All parameters are loaded to one variable accessible to the Simulink model. The parameters describe the main particulars and machinery data for the vessel in a given loading condition. In addition, 32 coefficients are included for use in Hollenbachs method for resistance prediction. The MATLAB code for loading vessel data is kept from the original simulation model by Bakke and Tenfjord (2017).

Engine data

Engine data is loaded by running a MATLAB function called *engine_pre.m*, which returns three parameters for each engine. The parameters must be configured manually in the mentioned function, which also is kept from the original simulation model. The following engine parameters are needed for running the simulation:

- Maximum continuous rating, MCR [kW]
- Brake-specific fuel consumption, BSFC [g/kWh]
- Optimal engine load [% of MCR]

5.1.2 Creating a new route

The ability to generate new routes is ensured by implementing the main functions from the RouteCreator script directly into the *run_sim* script. When creating a new route, the user will be asked to either select ports from the list of predefined alternatives, or to manually input coordinates for the desired ports. Alternatively, one can combine a port from the list and one of custom choice. The user must also define preferences for the maximum length for the sailing legs, and whether to include routes in polar waters. The process for setting up routes is similar as described in chapter 3.

After a route is generated, a function named *check_route* is run to validate the resulting route, and to determine if the process is to proceed to load weather data and run a simulation. This is done by visual inspection of the route on a map by the user. The function for checking the route only gives two options, either to keep the route and run a simulation, or to discard the route and terminate the process. There are thus no options to modify the resulting route or create a new route, as is present in the RouteCreator script. Such features are not included in the simulation algorithm in order to keep the code simple, and to keep focus on the main objective of running a simulation.

5.1.3 Met-ocean data

The wind and wave conditions can be described by several different parameters. Those used in the simulation model are listed in table 5.2. Wind speeds at 10 metre vertical height, are given as a northward($v10$) and eastward($u10$) component of the total wind speed. The mean wave direction denotes the direction the waves are propagating towards (oceanographic convention) and is given in degrees clockwise from true North. The significant wave height is the height of combined wind waves and swell.

The algorithm is made for wave and wind data stored in separate files, and two different MATLAB functions are therefore responsible for loading the met-ocean data. The functions have similar algorithms and differs only in the types of data loaded in each function. In addition to the parameters themselves, the latitude, longitude, and time of each data point is loaded to separate input variables. This is done for each file, allowing for differently defined data points for the wind and wave data file.

Prior to loading the data, both functions first check to verify that the met-ocean data covers the area of the given route and the given time span. If the area or time span is not covered by the supplied data, a warning will be displayed before an error will occur. The included met-ocean data covers latitudes from 70.20 to 3.10 degrees north, and longitudes from -180 degrees west to 179.70 degrees east, with a grid resolution of 0.3 degrees. The data covers dates from 1 January 2016 at 00:00, to 31 December 2017 at 23:00, with time intervals of one hour.

If the met-ocean data covers the given route and date, a proper subset of the data is extracted. This is done to reduce the size of the data loaded into MATLAB to be used as input parameters in Simulink. The different data parameters are loaded for an area covering the latitudes and longitudes spanned by the route, and for a time period constrained by the given start date and an estimated end date. The latter is calculated from time spent when sailing at 60 % of the set speed. It is necessary to load only the relevant data, not just to reduce the time used to run a simulation, but also since it impossible to load complete

met-ocean datasets of a certain size. Routes crossing the international date line in either direction, is accounted for when finding a suitable subset to extract. Including routes that first cross the date line in one direction and later in the opposite direction, when circumventing a land area.

In order to be able to load the met-ocean data for the route used in the presentation of the original simulation model, it necessary to reduce the size of the data. This is achieved by excluding some of the data points. The algorithm fist checks the resolution of the time intervals, as well as the resolution of the latitude and longitude grid, for which the data is given. If the data is given with time intervals of one hour or less, every other point in time is excluded from the dataset that is loaded. The same is done for latitudes and longitudes, if the grid resolution is less than 0.4 degrees and the number of entries needed to cover the route is above 400. The grid resolution of latitudes and longitudes are inspected separately, and data point can be excluded in any dimension independent of the other.

In order to handle dates efficiently in MATLAB, all dates are converted to a serial date number. In MATLAB this number is defined as the number of days from January 0 0000. The dates in the included met-ocean dataset is however given in hours from January 1, 1900. The times for each data point are therefore converted to comply with the format used by MATLAB. The met-ocean data files should be of netCDF format and must contain the variables listed in table 5.2. The longitudes for the data points need be organised in an ascending order. Furthermore, the parameter names in the dataset need to match those used in the code for the described functions. The code is included in appendices E.1.2 and E.1.2.

Table 5.2: Met-ocean data used in simulation model

Parameter	**Unit**
10 metre U wind component, $u10$	[m/s]
10 metre V wind component, $v10$	[m/s]
Significant wave height, H_s	[m]
Mean wave direction, mwd	[degrees]
Peak wave period, T_p	[s]

5.1.4 Output

The last part of the overall algorithm deals with processing of output data recorded during the simulation and generating plots. Several parameters are stored for each time step of the simulation. These include voyage data, wave and wind conditions, engine performance, and total resistance and resistance components during the voyage. Also, total time spent on the journey and total fuel consumption is calculated, along with average resistance and average load on main engine. All the output data is stored in a structure array variable, that can be saved in a MATLAB file with a preferred file name.

A separate script is executed to plot some of the results. And finally, a function generates two maps. One with an orthogonal projection, displaying the route. And one with Mercator projection, showing an animation of the simulated voyage.

Missing data

The met-ocean data can have entries where data is missing. At each time-step of the simulation, met-ocean data is gathered for the four entries nearest the current location, and the two closest points in time. A variable indication if data is missing, for all four locations and both points in time, is stored as a part of the simulation output. This is done for the wave and wind data separately. If the variable is equal to 1 for any time-step, the parameters stored as results for that time step is calculated with missing met-ocean data and is thus not reliable. Entries with missing data generally represent locations close to shore in the wave dataset.

Results calculated for points where met-ocean data is missing, is corrected for in the processing of the results. A specialised function is executed in such instances. The correction is performed by replacing the values of any time-step with missing met-ocean data, with the values of the previous time-step. This is done for all parameters, for all such time-steps. If the first time-step is based on missing data, the first entry of all stored parameter is replaced by the value of the next reliable entry. Is assumed that this is a reasonable correction given that few, if any, time-steps is expected to have missing met-ocean data. Also, small variations in weather conditions is expected from one timestep to the next, if relatively small time-steps is chosen in the simulation configuration. Not adjusting faulty entries could lead to misleading results.

5.2 The Simulink model

A description of the simulation model is given in the following. An overview of the Simulink model is shown in figure5.2. The model consists of an entity generator, an entity terminator, four entity servers, two entity switches, and a stop simulation block. The entity generator creates an entity representing the vessel, which flows through the other blocks of the model. Only one entity is used for the complete simulation. The input and output switches direct the flow of the vessel entity and enables the entity to loop through three entity servers. The loop with the three servers represents the iterative process of sailing from one point to the next along the route. The model also consists of seven Simulink functions, seen in the lower part of figure 5.2. These are called from different entity servers during a simulation run.

5.2.1 The simulation process

A flow chart of a simulation run is illustrated in figure 5.3. The different simulation subprocesses are shown in the rounded rectangles, while the simulation variables updated during a process are shown in the adjacent yellow parallelograms. The blue parallelograms display parameters stored as output during the connected process.

The simulation starts by generating a vessel entity, created with the 15 attributes listed in table 5.3. These are used to track and update the simulation variables as the simulated journey advances. Those listed with an initial value of 111, have initial values that are specific for a given simulation, and get their real initial value in later simulation processes. The value of 111, is mainly chosen to discern those attributes, from those whose values

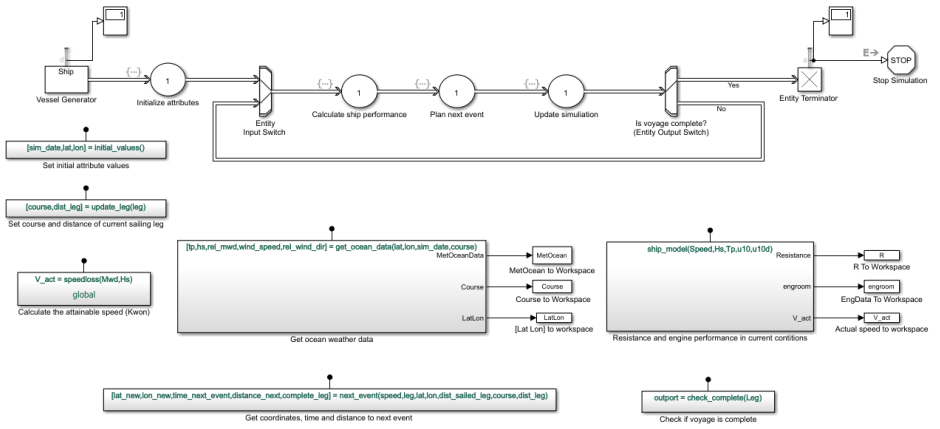


Figure 5.2: Simulink model

are universal, i.e. independent of the specific vessel and route to be simulated. The latter group keep their initial values set in the entity generator.

The next process assigns initial values to the simulation variables whose value must be set in order to start the simulated journey. This is done by assigning values to the corresponding entity attributes. In the Simulink model this task is performed by the first entity server, seen outside the sailing loop in figure 5.2. The simulation date is set to the given start date of the journey, while the latitude and longitude is set as the coordinates of the first waypoint of the route, representing the port of origin. The course and distance of the first leg is read from the route data.

The next step of the process retrieves met-ocean data for the current place and time. The variables returned are significant wave height, mean wave direction, wave period, wind direction and wind speed. The weather conditions, course, and coordinates of the current event is stored to the MATLAB workspace. The following sub-process then use the significant wave height and mean wave direction to estimate the speed loss due to waves. The attainable speed in the current conditions is estimated by applying a formula presented by Kwon (2008), later modified by Lu et al. (2015).

Thereafter the vessel performance parameters are calculated. The attainable speed and the significant wave height are used to calculate the calm water resistance and the added resistance due to waves. The resistance due to air and wind is found by using the attainable speed, the wind speed and the relative wind direction as input. The total resistance and attainable speed are used to calculate the brake power, fuel consumption and the engines operating point. The resistance, engine data and attainable speed at the current event are stored to the MATLAB workspace. The processes of fetching met-ocean data, estimating attainable speed, and calculating vessel performance parameters, are all performed by one entity server in the Simulink model.

After the vessel performance is calculated, a sub-process responsible for planning the next event, is executed. The distance to, and time until the next event is set, along with the coordinates of the next event. And the relevant simulation variables are updated. A

Table 5.3: Attributes of the ship entity

Attribute name	Description	Initial value
Sim_date	Simulation date [days from Jan. 0, 0000]	111
Lat	Current latitude [degrees North]	111
Lon	Current longitude [degrees East]	111
Leg	Current leg number	1
Course	Heading [degrees off North]	111
Att_speed	Estimated attainable speed [kn]	111
Distance_leg	Length of current leg [nm]	111
Distance_sailed_leg	Distance sailed of current leg [nm]	0
Distance_sailed_tot	Total distance sailed [nm]	0
Distance_next_event	Distance to position of the next event [nm]	0
Time_next_event	Time until next event at current speed [h]	0
Lat_next	Latitude of next event [degrees North]	111
Lon_next	Longitude of next event [degrees East]	111
Output_port	Output port for termination switch	2
Leg_completed	Indicator showing if leg is completed or not	0

preferred time step between simulation updates is defined in the simulation configuration, and the time between events is generally equal to this value. The distance to the next event is then set as the distance it is possible to sail in the given time step, given the attainable speed in the current conditions. The route is divided in sailing legs with a fixed distance. At the end of each leg, the distance it is possible to sail in the preferred time step, will therefore be longer than the remaining distance off the leg. In such instances the distance to the next event is set to the remaining distance of the leg. The time to next event is then calculated from the remaining distance and the attainable speed. The simulation variable indicating if a leg is completed is given a value of 1, if the next event marks the completion of a leg.

When the next event has been planned, the simulation is advanced to the next event. A task performed by the last entity server in the sailing loop in the Simulink model. The simulation date, latitude and longitude are updated to that of the next event. In addition, the distance sailed of the current leg and in total is updated. In the cases when the new event is not the end of a sailing leg, the simulation starts a new iteration of sailing by going back to the process of retrieving weather conditions for the current event. This iteration continues until a leg is completed.

When a leg is completed, the algorithm checks if the journey is also completed, by comparing the coordinates of the current event with those of the destination. If the destination has not yet been reached, the simulation starts a new leg by updating relevant simulation variables. The leg number is incremented, the course and distance are updated from the route data, and the simulation variable tracking the distance sailed of the current leg, is reset. Thereafter the simulation goes back to the process of retrieving weather conditions for the current event, and a new iterative process of sailing along the new leg has started.

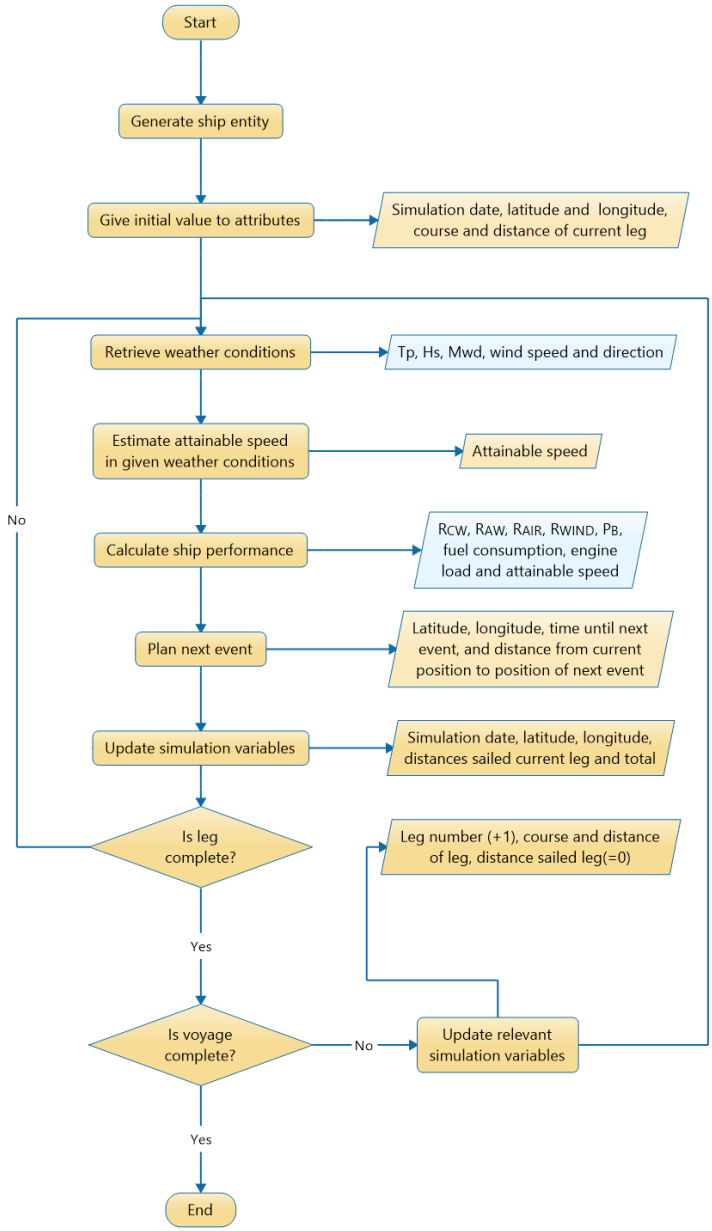


Figure 5.3: Flow chart of simulation model

The simulation continues until a leg of the route has been completed, and the coordinates of the new event corresponds to those of the destination port. At this point the simulated journey has reach its destination, and the simulation is terminated.

The Simulink blocks

The tasks performed by the different blocks of the model are summarised in the following. Most of the calculations are performed by MATLAB functions located in the different Simulink functions, which in turn are called from different entity servers to execute the desired calculations.

The first entity server assigns initial values to those attributes that must have set value prior to the entity entering the simulated sailing process. This is done by calling on two Simulink functions. The stating date and initial latitude and longitude, is set by the Simulink function specialised for the task, named *initial_values*. The initial course and distance of the leg is retrieved from the function called *update_leg*. A function utilised throughout the simulation when the simulated journey starts a new leg of the given route.

The first of the three entity servers in the sailing loop is responsible for calculating the vessel performance at the current event. The server calls on three different Simulink functions. First the relevant weather conditions are retrieved from the function named *get_ocean_data*. Thereafter the attainable speed in the current conditions is retrieved by calling the function named *speedloss*. The entity attribute for the attainable speed is updated at this point. When the weather conditions and the attainable speed have been determined, the ship performance is calculated by calling the function *ship_model*.

The planning of the next event is done when the ship entity enters the second entity server in the sailing loop. This block calls on the function named *next_event*, which is responsible for performing the necessary calculations, before the relevant attributes are updated.

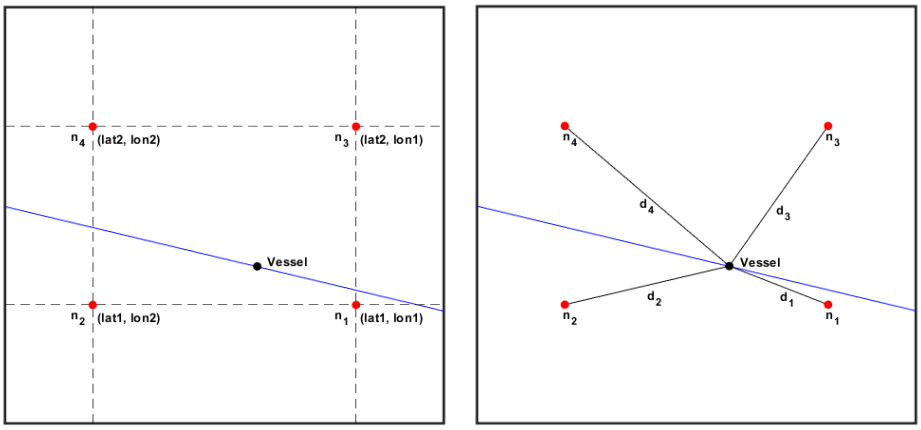
The final entity server is responsible for advancing the simulation by updating the relevant simulation variables to that of the next event. This is the only entity server with a set service time. The simulation clock is correlated to the simulated time of sailing, by setting the service time equal to the time till the next event. The *Leg_completed* attribute indicates if the current update marks the completion of a sailing leg. When a leg is completed, a Simulink function called *outport* is called, to check if the update also means the completion of the journey. The attribute *Output_port* is then updated with the value returned from this function. If the destination is not yet reached, a new leg is started by updating the relevant simulation variables. This includes calling on the *update_leg* function, to get the course and distance of the new leg.

After all relevant attributes have been updated, the entity moves to the output switch. Here the next step of the entity is determined by the *Output_port* port attribute. The entity is then either directed back to the first entity server in the sailing loop, or to the entity terminator if the journey is completed. The *stop simulation* block ends the simulation when an entity is registered to have arrived at the entity terminator.

5.2.2 Retrieving met-ocean data

The met-ocean data is given in a gridded format, meaning that the data is available for coordinates given by a grid of latitudes and longitudes with a fixed resolution. The data is also given for a certain time span, with a certain time interval between the data points. A specific latitude, longitude, time and date is thus needed to read the values of a given met-ocean data parameter. Barring random chance, the coordinates of the vessel at any

given event will not match the coordinates of the met-ocean data. The same is true for the time and date at any event. The met-ocean data at each event is therefore determined by interpolating data from several data points. Generally, eighth data points are used, representing the four point in space nearest the current position, and two points in time nearest the current simulation date. The interpolation is performed by summarising the parameter values from the different data points, while applying a relative weighting to each of the four sets of coordinates, and the two points in time. Values are determined for the parameters listed in table 5.2.



(a) The four met-ocean data points with closest coordinates to the vessel position

(b) Distances to each data point, used in weighting the individual parameter values

Figure 5.4: Locations of met-ocean data points used in interpolation

The first step for retrieving met-ocean data is to identify the two points in time in the dataset, t_1 and t_2, closest to the simulation date, t_{sim}. The next step is to identify the two latitudes and the two longitudes, in the dataset, closest to the position of the vessel at the current time-step. This gives four data points, n_1 - n_4, located at the nearest distance to the position of the vessel, as illustrated in figure 5.4a. Now the parameter values at each location and time can be read. As an example, $H_s(t_1, n_3)$ is the value of the significant wave height at time t_1 and location n_3, with coordinates $(lat2, lon1)$.

A weighting of each of the two points in time is found based on the linear distance in time between t_{sim} and t_1 and t_2, respectively. The weighting is calculated as shown in equation 5.1, where the relative weighting time point t_1 is found by letting $i = 1$.

$$w_{t_i} = 1 - \frac{|t_{sim} - t_i|}{|t_2 - t_1|}, \qquad i = 1, 2 \tag{5.1}$$

The distances from the nearest locations to the position of the vessel can be seen in figure 5.4b. A relative weighting of each point is calculated as the inverse ratio between the distance, d_j, from the vessel to each point, and the total distance D. A normalised weighting is obtained by dividing the result by the sum of the inverse ratio for all distances.

The formula is displayed in equation 5.2, where N is the number of nodes used in the interpolation, and the total distance is given as $D = \sum d_j$.

$$w_{n_j} = \frac{\frac{D}{d_j}}{\sum_{k=1}^{N} \frac{D}{d_k}}, \qquad j \leq N, \quad j, N = 1, 2, 3, 4 \tag{5.2}$$

When the relevant data points and their relative weighting have been found, the interpolation of the parameter value is performed as shown in equation 5.3, where the significant wave height H_s is used as an example.

$$H_s(t_{sim}, lat, lon) = \sum_{i=1}^{2} \sum_{j=1}^{N} H_s(t_i, n_j) w_{n_j} w_{t_i}, \quad N = 1, 2, 3, 4 \tag{5.3}$$

Missing data

Data points where met-ocean data is missing are excluded from the interpolation, in order to avoid erroneous parameter values. At locations where data is missing, the parameter values are given as *NaN*, meaning "Not a Number". Missing data is thus found by searching the relevant data points for NaN values. It is assumed that all parameters are missing if one is missing for a given data point. Therefore, only one variable is checked for each dataset. Arbitrarily chosen, Tp and $U10$ is used for this purpose.

Each point in time is checked for missing data separately. If both points in time have valid data from at least one node, the parameter values are found as described above. Should one point in time have missing data for all the nearest data points, only data from the other point in time is used. The parameter values are then found by interpolating the values at nodes with valid data for the one point in time.

There is a possibility that all four the nearest data points have missing data for both points in time. As previously described, such instances are dealt with in the processing of the simulation results. For this purpose, a variable indication if data is missing for each event is stored and sent to the MATLAB workspace. This is done for each dataset.

Met-ocean parameters

The task of the *get_ocean_data* function is to establish suitable values for met-ocean parameters used to evaluate the vessel performance. The parameters returned from the function are listed in table 5.4. The peak wave period and the significant wave height are returned with the values found from the interpolation of the met-ocean data. The peak wave period is however not in use in the current version of the model. It is included since the original model had it included, for performing resistance calculations with *ShipX*.

The resultant wind speed is calculated from the eastward and northward components, $u10$ and $v10$, as shown in equations 5.4. The direction of the resultant wind speed can be found from equation 5.5. The wind direction, θ, as calculated with equation 5.5, is the direction the wind is blowing towards in degrees from the North meridian. Furthermore, the wind direction relative to the ship heading, β, is calculated according to equation5.6. The relative wind direction, θ_{rel}, is then the direction the wind is coming from, in degrees

clockwise from the centre line. Thus, head wind gives $\theta_{rel} = 0°$, and side wind from port side yields $\theta_{rel} = 270°$.

The relative wave direction is calculated as the angle of the incoming waves relative to the ship heading. Equation 5.7 yields the relative wave direction in degrees between 0 and 180. Head waves is then given as $mwd_{rel} = 0°$, and waves from the stern as $mwd_{rel} = 180°$.

$$U_{res} = \sqrt{u10^2 + v10^2} \tag{5.4}$$

$$\theta = 90\Big(2 - \frac{u10}{|u10|}\Big) - \arctan\Big(\frac{v10}{u10}\Big)\Big(\frac{180}{\pi}\Big) \tag{5.5}$$

$$\theta_{rel} = \theta - \beta + 180c, \quad c = \begin{cases} -1, & \text{if } (\theta - H) > 180 \\ 3, & \text{if } (\theta - H) < -180 \\ 1, & \text{otherwise} \end{cases} \tag{5.6}$$

$$mwd_{rel} = \begin{cases} |mwd - \beta| - 180, & \text{if } |mwd - \beta| > 180 \\ 180 - |mwd - \beta|, & \text{otherwise} \end{cases} \tag{5.7}$$

Table 5.4: Met-ocean returned from *get_ocean_data* function

Parameter	**Unit**
Resultant wind speed (U_{res})	[m/s]
Wind direction relative to heading (θ_{rel})	[degrees]
Significant wave height (H_s)	[m]
Relative mean wave direction (mwd_{rel})	[degrees]
Peak wave period (T_p)	[s]

5.2.3 Vessel performance calculations

A brief presentation of the vessel performance calculations in the Simulink model is given in the following.

Speed loss estimate

The method and code for estimating the speed loss in different weather conditions are kept as described for the original simulation model by Bakke and Tenfjord (2017). The method used is a formulae for speed loss approximation by Kwon (2008), as modified and presented by Lu et al. (2015). The estimate is based on the parameters listed in table 5.5. In addition to the listed parameters, the speed loss formula also depends on loading condition and type of vessel(Lu et al., 2015). The algorithm must therefore be customised for each specific vessel design and loading condition.

The Beaufort Scale categorises wind speeds and sea states, so that sea conditions can be describes by the Beaufort Number(Garrison, 2005). In the model, the Beaufort Number

is determined from the significant wave height. Interestingly, the Beaufort Number is set equal to seven for all significant wave heights greater than four metre, even though the scale continues. The model also use volume displacement in the speed loss estimate, where displacement tonnage is used in the paper by Lu et al. (2015).

Table 5.5: Parameters used to estimate speed loss

Description	Unit
Design speed (V)	$[m/s]$
Beaufort Number (BN)	[-]
Froude Number (F_n)	[-]
Block coefficient (C_B)	[-]
Relative mean wave direction (mwd_{rel})	[degrees]
Volume displacement (∇)	$[m^3]$

The ship model

The ship model is tasked with calculating the vessel performance, and is made up of four blocks, as can be seen in figure 5.5. All blocks are kept from the original simulation model, as described by Bakke and Tenfjord (2017). Only the code for calculating the air and wind resistance, has been modified.

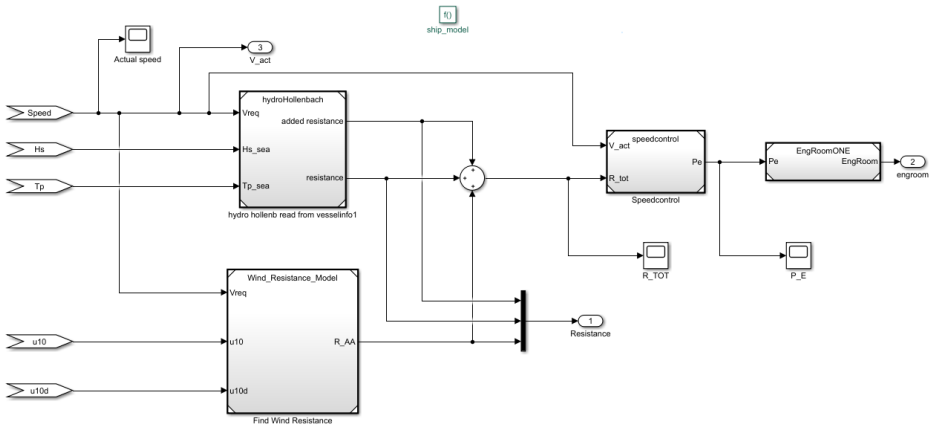


Figure 5.5: The ship model in Simulink

When the Simulink function is executed, two different blocks first calculates the hydrodynamic resistance, and air resistance, respectively. The hydrodynamic resistance is estimated based on Hollenbach's method, modified to include added resistance in waves, with the significant wave height and attainable speed as input variables. The air resistance is calculated to account for resistance due to wind, according to equation 5.8, where C_{air}

is the air resistance coefficient for the superstructure, and A_p is the transverse projected area of the superstructure(Steen and Minsaas, 2014).

$$R_{AA} = C_{air} \cdot \frac{\rho_{air}}{2} \cdot U_{rel}|U_{rel}| \cdot A_p \tag{5.8}$$

U_{rel} is the wind velocity relative to the vessel speed, V, and is calculated according to equation 5.9. The latter part of equation 5.9 express the component of the wind velocity in the direction of the ship heading, such that the velocity of head wind component is positive and tail wind component is negative.

$$U_{rel} = V + U_{res} \cdot \cos(\theta_{rel})\Big(\frac{180}{\pi}\Big) \tag{5.9}$$

After the resistance components have been determines, a third block calculates the towing power needed, from the total resistance and the attainable speed. Finally, a block representing the engine room, calculates the brake power, the relative engine load and the specific fuel consumption. The engine room block included in the current model represents a single engine. It can however be replaced by other blocks, also supplied by the original model, representing different engine configurations.

5.2.4 Comparison to the original model

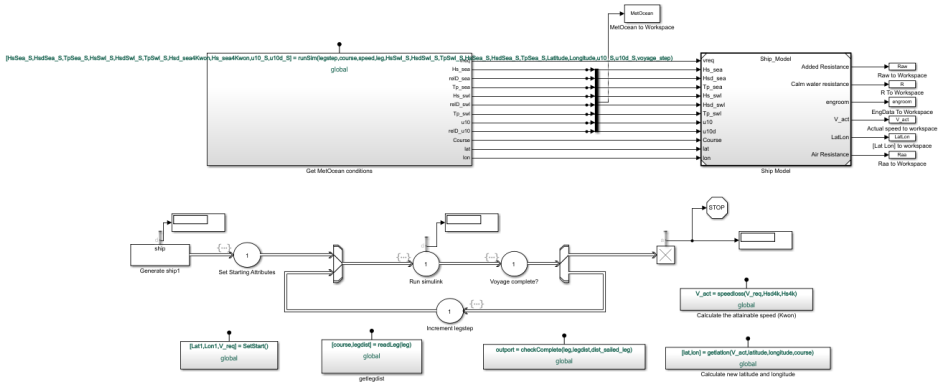
The code and functions for vessel performance calculations are mostly kept from the original simulation model by Bakke and Tenfjord (2017). It is therefore useful to compare the current simulation model with the original. The following highlights some differences between the models.

The greatest difference lies in how the models incorporate the met-ocean data. The original model use Markov chains to model the wind and wave conditions, with several states defined for each met-ocean parameter. Markov chains include using transition probabilities to determine the next state of a system based on the current state, in a stochastic manner. The original model also depends on a predetermined route with a fixed number of legs, and with met-ocean data as input for each leg. This requires the met-ocean data to be pre-processed for each route to be simulated. The current model uses hindcast met-ocean data directly, removing the need to pre-process the data according to each route. By doing so, time must be included as a simulation variable. In the original model, time was only included by setting a fixed time step between updates.

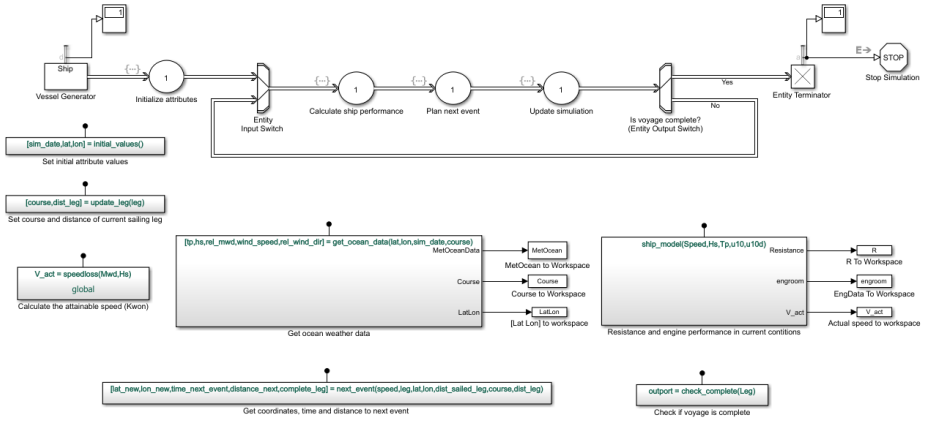
Another difference regarding the met-ocean data is that the current model uses wave data for combined swell and wind waves, while the original model uses separate wave data for the two. This change is merely based on the available data, and not intended to improve the model. In the original model, the added resistance in waves is calculated separately for swell and wind waves, when using the ShipX module for resistance calculations. But when using Hollenbach's method, the wave data is combined prior to resistance calculations. The current model is created for using Hollenbach's method and must be modified in order to enable using the ShipX module.

A less important difference between the models lies in the conventions used for some of the met-ocean data input. In the original model, the wind is described by a wind direction in nautical conventions, and resultant wind speed at 10 metre height. In nautical

convention, the direction describes the angle where the wind is blowing from, such that $0°$ denotes wind from North and $90°$ denotes wind from the East. The current model use wind vectors as input. Also, the original model use mwd in nautical convention, while the current model use mwd given in oceanographic conventions.



(a) The original model



(b) The current model

Figure 5.6: A comparison of the two Simulink models

The ship model is the Simulink block responsible for calculating the vessel performance parameters at each step of the simulation. In the original model this block is activated when the Simulink function, responsible for determining the values of the met-ocean parameters, is called from the entity server named *Run simulink*. The ship model of the original simulation model can be seen in the top right corner of figure 5.6a. The chosen configuration of the original model included replicating the function for calculating the attainable speed, and the function for finding the coordinates of the next step of the journey. Both functions are executed twice for each step of the simulation.

The speed loss is first calculated in the ship model, by using Hs, from combined

waves, and mwd for wind waves only. The value calculated here is stored to the MATLAB workspace. It is also used to calculate coordinates of the next step, also stored to the workspace. When the met-ocean data parameters are returned to the entity server, another Simulink function is used to calculate the attainable speed once more. This time both Hs and mwd for wind waves only is used as input. The value calculated at this point is used as a simulation variable for the attainable speed and is also used to calculate coordinates for the next step of the simulation. These coordinates are used as simulation variables for the current latitude and longitude and is thus used for calculation met-ocean parameters at the next simulation update.

Creating a new function for reading met-ocean data yielded an opportunity for creating a more intuitive flow of the simulation model, dividing the simulation of sailing into three processes and removing the redundant functions. In the current model the ship model is made as a Simulink function executed from the first entity server in the sailing loop. Furthermore, the attainable speed, and the coordinates of the next event, is only calculated once each time step. Both Hs and mwd for combined swell and wind waves, are used in the calculations.

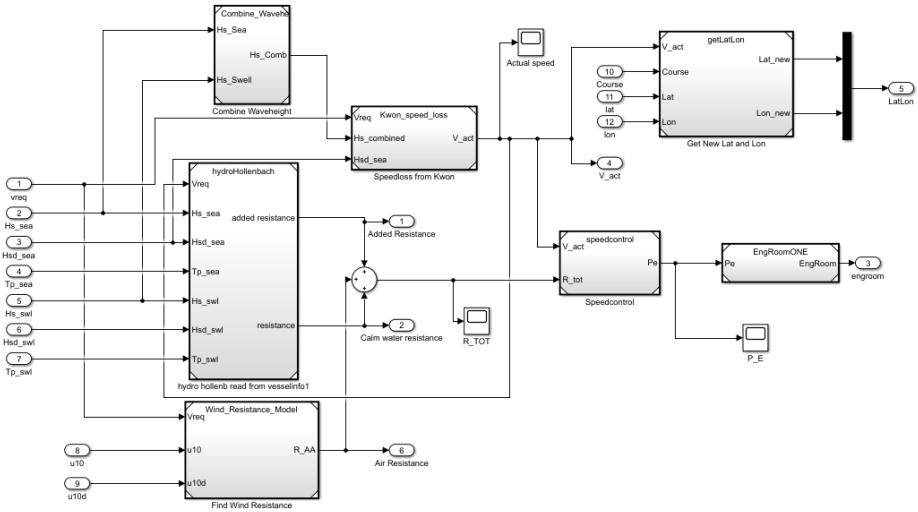


Figure 5.7: Ship module from the original model

The blocks used in the new ship model is kept from the original ship model, but the redundant blocks have been removed in the current version. This can be seen by comparing figures 5.5 and 5.7. The original model uses attainable speed as input for Hollenbach's method, but design speed when calculating the wind resistance, while the current model use attainable speed as input for both. Otherwise, only the code for air resistance calculations have been is altered. In the original model the air resistance is calculated as the change in air resistance due to wind (Steen and Minsaas, 2014). In the current version this is corrected to calculate the resistance from air and wind combined, as described in the last section.

Chapter 6

Results

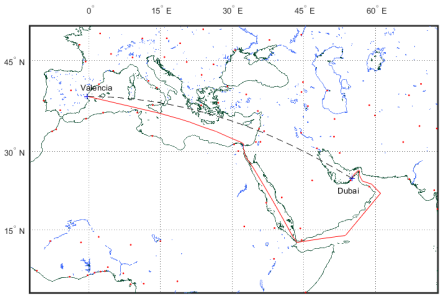
Results produced from the algorithm for automatic route-finding and results produced by the simulation model, is presented in this chapter. All results have been produced using MATLAB version R2017b on a Widows 10 laptop, with a 2 GHz AMD processor, an integrated AMD Radeon R5 Graphics processor and 12 GB RAM. First some results from running the RouteCreator script is presented in section 6.1. Then three cases are described in section 6.2, along with the results from running the simulation model for the case studies.

6.1 Results from the RouteCreator script

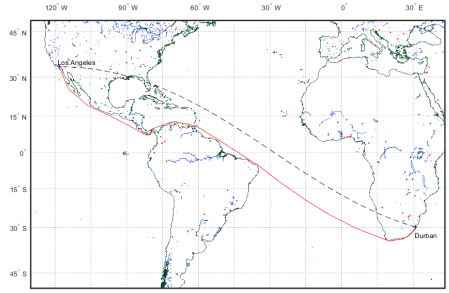
The RouteCreator script have been executed for all combinations of the 20 predefined ports in the included port database. The results, first when restricting access to polar waters, then without restrictions, are presented in section 6.1.1. The feature for setting up routes between manually defined ports has also been tested, and the results are presented in section 6.1.2. Finally the ability to to improve routes with the modification option is demonstrated in section 6.1.3.

The model creates identical routes between two ports, irrespective of which port is designated origin and destination port. To avoid redundancy, the results therefore only includes one route for each combination of ports. The RouteCreator script ran repeatedly without any execution errors. A selection of the results is presented in the following, while all results are included in the electronic appendix.

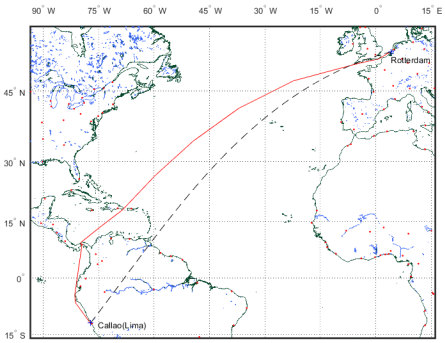
Special knowledge is required to assess the feasibility and optimally of a given route. Lacking such expertise, the resulting routes are assessed based on a shortest path perspective, where an approximate shortest path route is deemed acceptable. Any clearly non shortest path routes, and routes with unnatural sharp turning points, are assessed as non-acceptable.



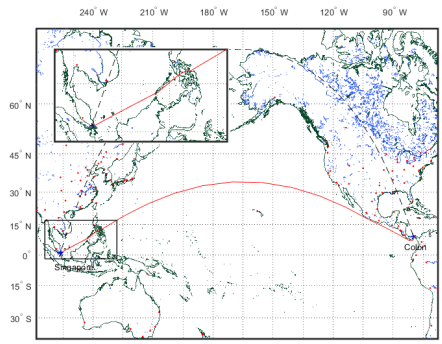
(a) Route between Dubai and Valencia



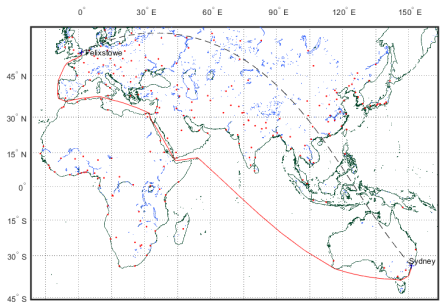
(b) Route between Durban and Los Angeles



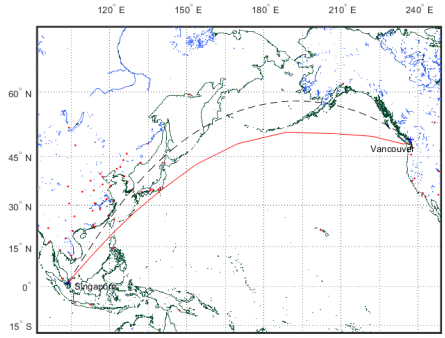
(c) Route between Callao (Lima) and Rotterdam



(d) Route between Colón and Singapore



(e) Route between Felixstowe and Sydney



(f) Route between Singapore and Vancouver

Figure 6.1: Examples of routes deemed acceptable, generated with the RouteCreator script. Maximum length of sailing legs set at 800 nm

6.1.1 Routes between the pre-defined ports

Creating routes between all the included ports resulted in 190 unique routes, for each setting of maximum length of sailing legs and preference for polar sailing. Three different leg lengths have been used, 400 nm, 800 nm and 1200 nm. The results show 217 unique

routes for each sailing leg length, when excluding identical routes from each preference for polar sailing. In total 651 different routes have been created and inspected. The results presented in the following are for a maximum leg length of 800 nm. Similar results were achieved for the two other length choices. For the 190 routes created when restricting access to polar waters, it took on average 12.5 seconds to generate a route without displaying maps. The minimum and maximum time used were 1.8 seconds and 73 seconds.

Routes restricted from polar waters

Table 6.1 presents an overview of the results with sailing leg length of 800 nm and polar water access restricted. 71 of the 190 unique routes created, was deemed non-optimal. Of the non-optimal routes, ten routes had two issues making the route non-optimal, while 61 were made non-optimal by a single error. This is of importance since the script has an option to modify the resulting routes to make them acceptable. Any route deemed non-acceptable due to a single error can be fixed by the modification option, whilst this cannot be guaranteed for routes made non-acceptable from two or more errors. Thus ten of the 190 routes may not be sufficiently improved by the modification options.

A selection of the acceptable routes are displayed figure 6.1. The routes between Singapore and Sydney, Busan and Sydney, and all routes between New York and any of the ports on the American continents, were all strictly speaking non-feasible. The mentioned routes intersect one or several tiny islands near the shore of a larger island. These routes are still deemed acceptable, due to the small deviation needed to avoid the intersected islands. The described errors occur from shoreline sailing path polygons not being modified to describe navigable paths, and when the shoreline polygons for the intersected island are missing from the set of shoreline polygons used in the route-finding algorithm. As an example, the route between New York and Colón, Panama is shown in figure 6.2.

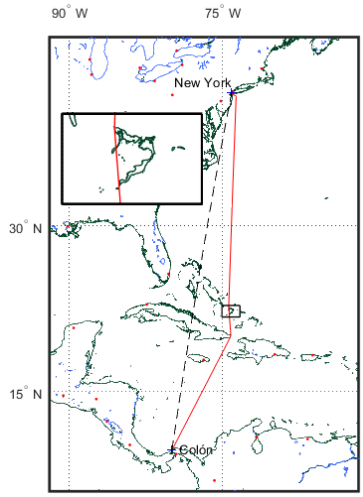


Figure 6.2: Route between Colon, Panama and New York intersects a small island in the Caribbean ocean

Table 6.1: Results when generating routes between all ports included in the port list, for max leg length of 800 nm and with polar sailing restricted.

Routes between the included ports	
Unique routes	190
Acceptable routes	119
Non-acceptable routes	71
Single error routes	61
Double error routes	10
Total number of error	81
Type-1 errors	61
Type-2 errors	11
Other errors	9

Figure 6.3 presents some non-optimal routes. The majority of the non-optimal routes are due to a non-optimal connection node being set for one or both ports. In table 6.1 and henceforth, this is referred to as a *type-1* error. Examples are illustrated in figures 6.3a and 6.3b. The first figure shows a route with an abrupt and unfeasible turning point, and the second figure displays a route with two type-1 errors. In most cases type-1 error are caused by an unfortunate location of the best available connection node, relative to the port location. Usually for connecting to the shoreline sailing path, around the land area where the port in question is located.

The second most prevalent error contributing to non-optimal routes, are sub-optimal selections of the order in which the land areas are being circumvented. Such *type-2* errors make up 11 of the total 81 errors behind the non-optimal routes. Examples are shown in figures 6.3c and 6.3d. The remaining nine errors are of varying nature. All issues resulting in non-optimal routes are listed below.

- **Type 1 errors:**
  - Routes from Valencia through the Strait of Gibraltar are non-optimal due to location of nearest visible connection node in relationship to the port.
  - Routes from Lagos to or through the Panama Canal have non-optimal connection-node for Lagos.
  - Routes from Hamburg towards or through the English canal have non-optimal connection node for Hamburg.
  - All routes from Tokyo, southwards to ports on the Eurasian continent have non-optimal connection nodes for Tokyo.
  - Routes southward from Shanghai have non-optimal connection nodes, due to the location of the nearest visible connection node in relationship to the pre-defined port path.
  - Route between New York and to or through the English canal have non-optimal connection nodes for New York.

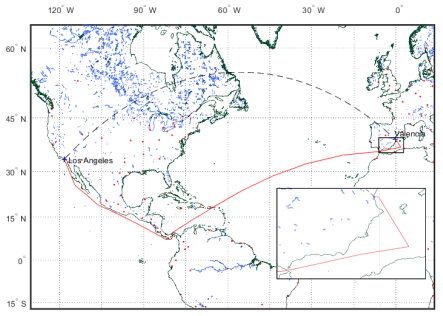
- **Type 2 errors:**
  - Routes from Busan to Lagos, Santos and Durban goes through the Strait of Malacca, when a shorter route is through the Sunda Strait.
  - Routes from Durban to ports in Northern Europe goes through the Suez Canal.
  - Routes between Shanghai and Tokyo goes through the Kanmon Straits, when it is shorter to sail south of the Japanese island of Kyushu.
- **Other errors:**
  - Routes from Marseilles eastwards through the Suez Canal toward ports on the Eurasian continent, yields an unfeasible route in the Mediterranean Ocean, due to non-optimal choice of connection node and then further disturbed by the visibility check
  - Route between Dubai and Los Angeles goes westwards from Dubai through both Suez and Panama, due to the nature of the great circle route between them. An eastward route is shorter
  - Routes between Mumbai and Dubai are longer than necessary due to the pre-defined ports paths
  - Route between Dubai and Lagos goes through the Suez Canal, when it is shorter to sail south of Africa

Comparison to online service

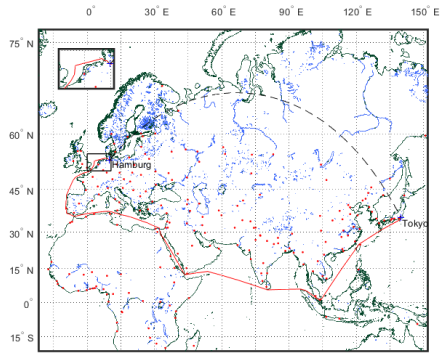
A comparison has been made between six acceptable routes made with the route-finding algorithm, and six routes between the same ports created with an online route-generating tool. The website *searoutes.com* offers a service for creating routes between ports based on routes extracted from AIS data (Searoutes.com, 2020). A comparison of the distances on the selected routes are presented in table 6.2. There is a good match of the distances in the small sample of routes, with differences in route distance varying between 0.08 % and 0.5 %. A visual comparison also confirmed a good match for the selected routes.

Table 6.2: Distance of six acceptable routes compared to distance of routes from searoutes.com

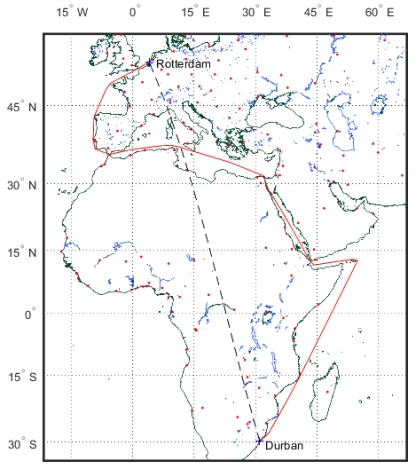
Route	searoutes.com	RouteCreator	Deviation
Mumbai - Valencia	4723 nm	4727 nm	0.08 %
Valencia - New York	3590 nm	3584 nm	0.17 %
Felixstowe - Sydney	11635 nm	11608 nm	0.23 %
Singapore - Vancouver	7102 nm	7111 nm	0.13 %
Callao - Rotterdam	6203 nm	6172 nm	0.5 %
Durban - Los Angeles	11178 nm	10170 nm	0.08 %



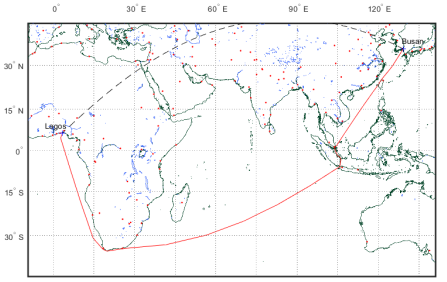
(a) Route between Los Angeles and Valencia



(b) Route between Hamburg and Tokyo



(c) Route between Durban and Rotterdam



(d) Route between Busan and Lagos

Figure 6.3: Examples of non-optimal routes. Polar sailing restricted, and maximum length of sailing legs set at 800 nm

Routes in polar waters

Running the RouteCreator script for 190 unique port combinations while allowing for routes in polar waters, resulted in 27 new routes. The reaming 163 routes were identical to the routes created when restricting access to polar waters. The results for routes in polar waters are summarised in table 6.3.

12 of the 27 routes were deemed acceptable. Figures 6.4a and 6.4b illustrates two such routes, through the North East Passage and North West Passage respectively. The two routes circumventing Antarctica were both evaluated as acceptable. Figure 6.5 displays juxtapositions of the restricted and non-restricted routes through antarctic waters.

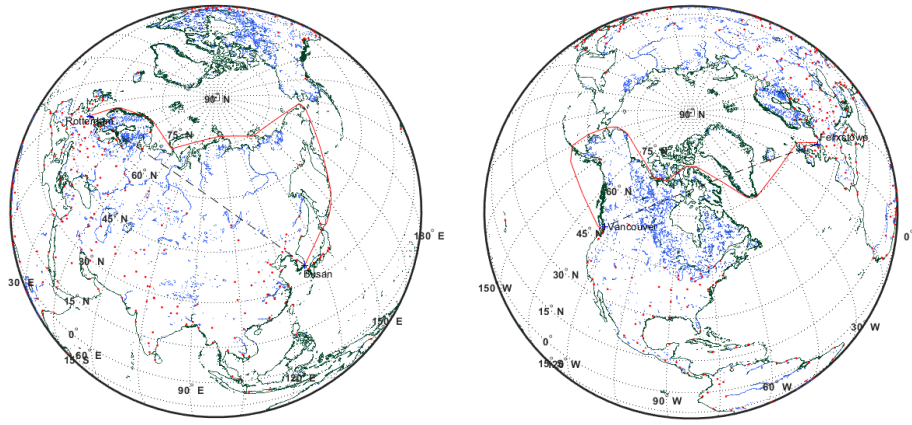
15 routes were made non-optimal by a total of 17 errors. Two routes, those between New York and Tokyo and between Tokyo and Valencia, suffered two errors for each route.

The remaining 13 routes were made non-optimal by a single error and are thus redeemable by the modification option, and 11 of the 14 errors were of type 1. Examples are shown in figures 6.6a and 6.6b. The type-1 errors resulting in non-optimal routes are listed below.

- **Type 1 errors:**
  - Routes from New York to Busan, Shanghai and Tokyo has non-optimal connection node for New York, due to location of nearest connection node in relationship to the port/ port path.
  - All six routes to or from Tokyo has non-optimal connection node for Tokyo, due to the nature of the algorithm selecting connection nodes when the port is located inside the shoreline sailing path polygon.
  - Routes from Valencia through the Strait of Gibraltar are non-optimal due to location of nearest visible connection node in relationship to the port / port path

Routes from Hamburg, Rotterdam and Felixstowe to Los-Angeles all suffers a type-2 error yielding a route through the North-west passage, when a route through the Panama Canal is both shorter, and not to mention more feasible. The route between Los Angeles and Rotterdam is illustrated in figure 6.6c.

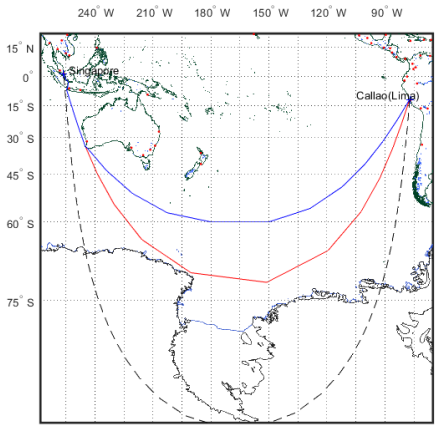
The remaining three errors occurred on the routes between Sydney and Felixstowe, Hamburg and Rotterdam. These routes were both non-optimal and non-feasible from intersecting land areas in the Solomon Islands. The error occurs due to missing shoreline polygons and shoreline sailing path polygons not being properly defined in the region. The route between Felixstowe and Sydney is displayed in figure 6.6d.



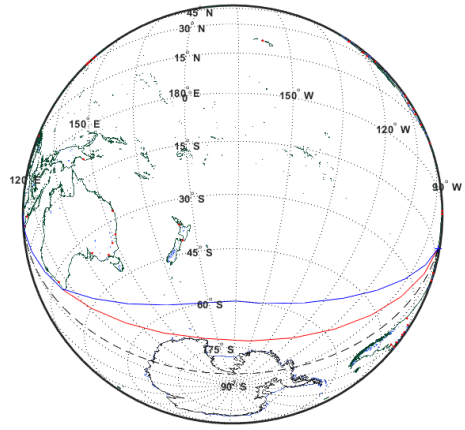
(a) Route between Busan and Rotterdam

(b) Route between Felixstowe and Vancouver

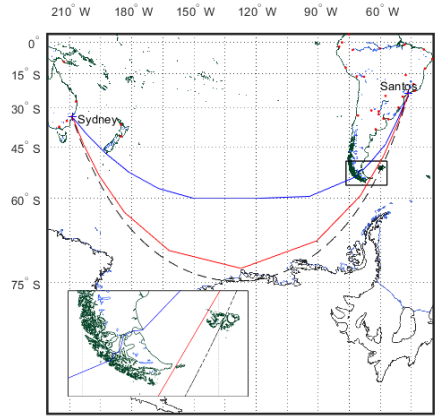
Figure 6.4: Two of the resulting routes going through arctic waters, deemed acceptable



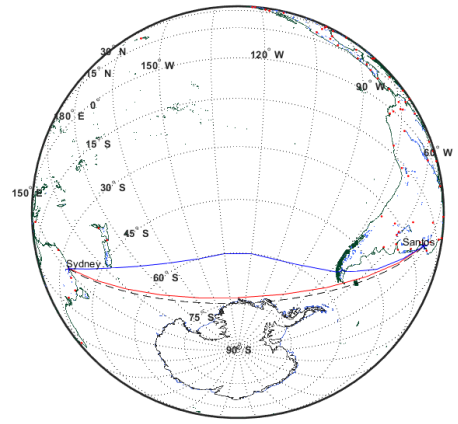
(a) Routes between Callao and Singapore



(b) Routes between Callao and Singapore, orthogonal projection

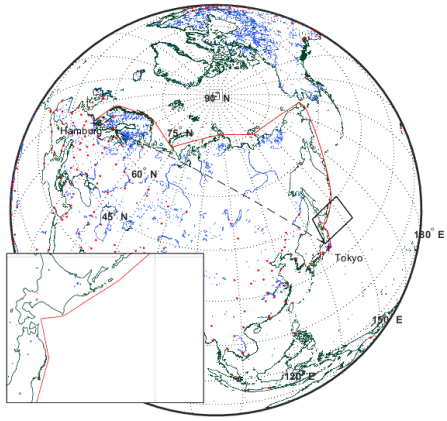


(c) Routes between Santos and Sydney

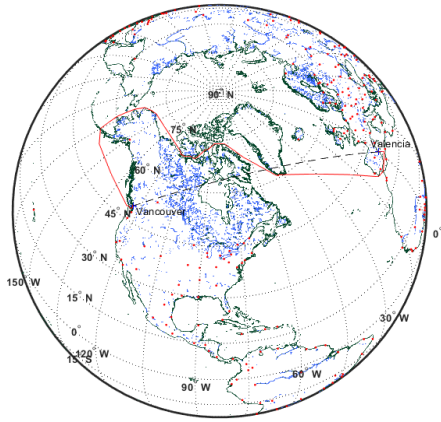


(d) Routes between Santos and Sydney, orthogonal projection

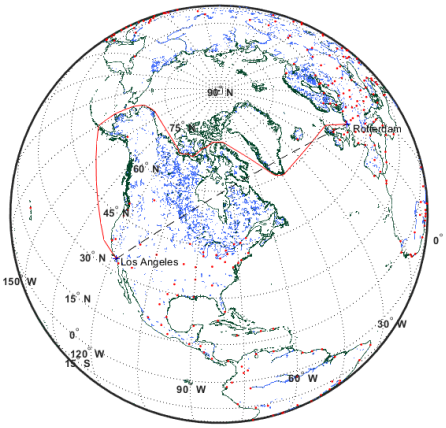
Figure 6.5: The two resulting routes through Antarctic waters. The blue line shows the routes with polar access restricted, and the red line show the resulting routes without restrictions.



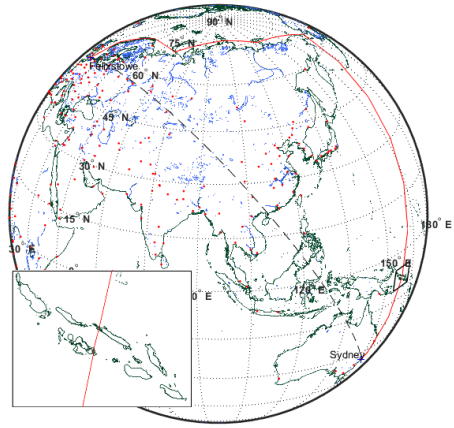
(a) Route between Hamburg and Tokyo



(b) Route between Valencia and Vancouver

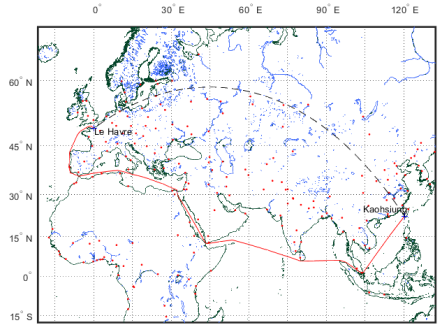


(c) Route between Los Angeles and Rotterdam

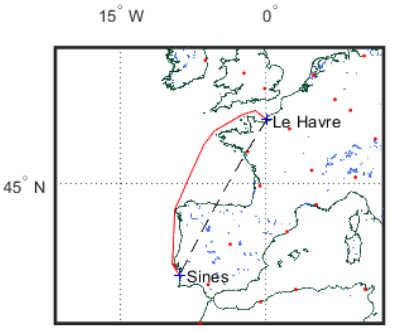


(d) Route between Felixstowe and Sydney

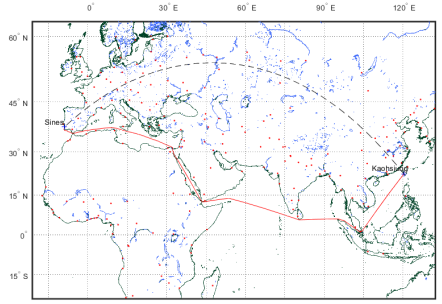
Figure 6.6: Some of the non acceptable routes through arctic waters



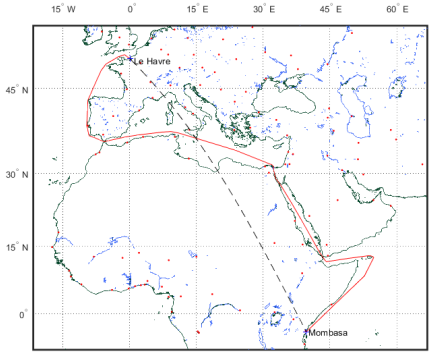
(a) Route between Kaohsiung, Taiwan and Le Havre, France



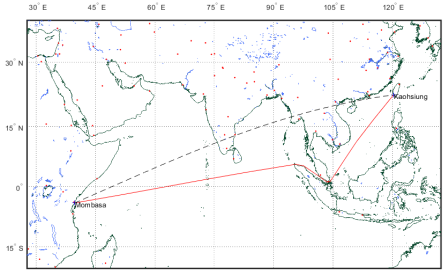
(b) Route between Le Havre, France and Sines, Portugal



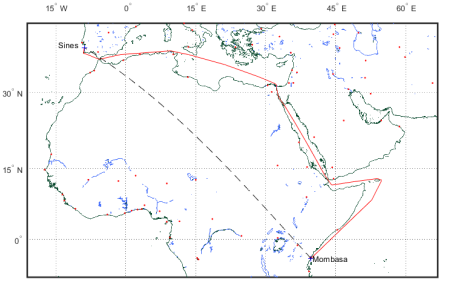
(c) Route between Kaohsiung, Taiwan and Sines, Portugal



(d) Route between Le Havre, France and Mombasa, Kenya

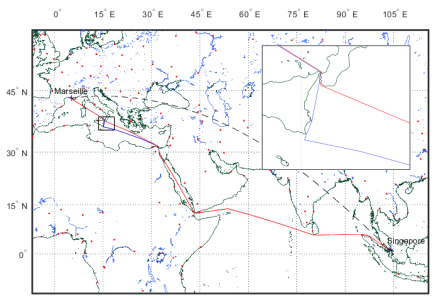


(e) Route between Kaohsiung, Taiwan and Mombasa, Kenya

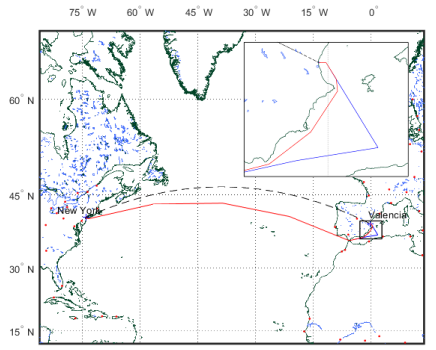


(f) Route between Mombasa, Kenya and Sines, Portugal

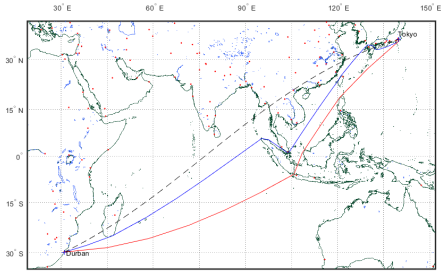
Figure 6.7: Routes between custom ports, from using the manual input option



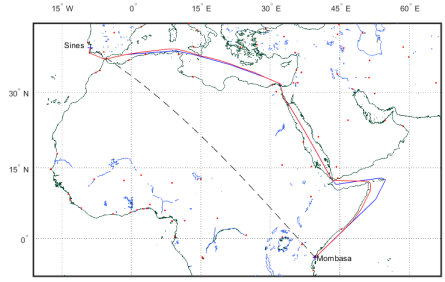
(a) Route between Marseille and Singapore



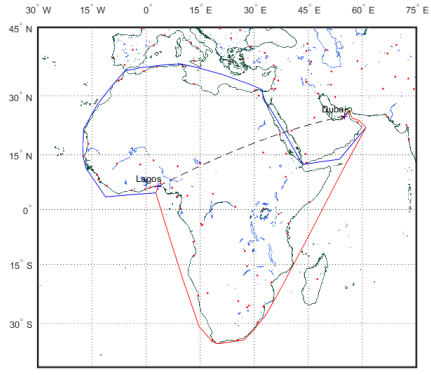
(b) Routes between New York and Valencia



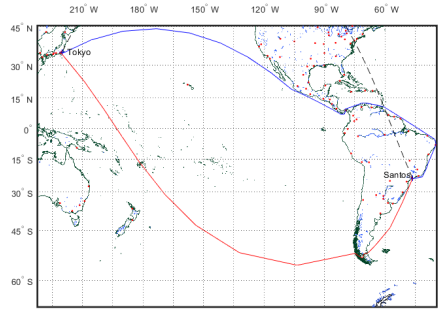
(c) Routes between Durban and Tokyo



(d) Routes between Mombasa and Sines



(e) Routes between Dubai and Lagos



(f) Routes between Santos and Tokyo

Figure 6.8: Original and modified routes, shown in blue and red respectively.

6.1.2 Routes between custom ports

The ability to generate routes between custom ports have been tested by running the RouteCreator script, while using the manual input option for defining the ports. Six routes have been created between the four following ports.

- Kaohsiung, Taiwan
- Le Havre, France
- Mombasa, Kenya
- Sines, Portugal

The results displayed in figure 6.7, are from using a maximum leg length of 750 nm. The routes from Mombasa to Le Havre and Sines are shown in figures 6.7d and 6.7f. Both routes are non-optimal when rounding the Horn of Africa, seen from a shortest path perspective. Other considerations like potential pirate activity is not considered, as previously mentioned. The remaining four routes were all acceptable.

Table 6.3: Results when generating routes between all ports included in the port list, when allowing for routes in polar waters and excluding identical routes. Max leg length of 800 nm.

Routes in polar waters	
Unique routes	27
Acceptable routes	12
Non-acceptable routes	15
Single error routes	13
Double error routes	2
Total number of errors	17
Type-1 errors	11
Type-2 errors	3
Other errors	3

6.1.3 Modifying routes

Figure 6.8 displays six routes where the modification option have been used to improve the original results. The original routes are shown in blue while the modified routes are shown in red. Only single error routes were modified.

The route between Santos and Tokyo, as seen in figure 6.8f, was modified by altering the maximum length of the sailing legs from 800 nm to 1200 nm. The original route goes through the Panama Canal and has a total distance of 12171 nm. The modified route goes through the Strait of Magellan, in the southern path of Chile, and has a total distance of 11369 nm.

The five other routes were modified by defining either a single waypoint or a path of waypoints, that the route must include. All routes were improved by the modification option, resulting in both shorter distances and eliminating strange paths and unnatural turning points. The distances of the original and modified results are listed in table 6.4.

Table 6.4: Distances of the original resulting routes and the modified routes.

Route	Original result	Modified result
Dubai - Lagos	8211 nm	7486 nm
Durban - Tokyo	7919 nm	7611 nm
Marseille - Singapore	6645 nm	6559 nm
Mombasa - Sines	5539 nm	5183 nm
New York - Valencia	3712 nm	3584 nm
Santos - Tokyo	12171 nm	11369 nm

6.2 Simulation results

A good approach to validate the simulation model would be to compare the results of a simulated voyage, with real world test data for the same vessel on an identical route. Lacking such test data, three case studies have been performed to gain insight on how the model responds in various conditions. In addition to the presented cases, the ability for setting up routes when running the simulation model has been validated with routes between several of the included ports.

First a case that looks at how the model reflects seasonal variations in weather conditions throughout a year, is presented. The second case is a simulated crossing between Rotterdam and New York, intended to shed light on the impact of the weather conditions on the vessel performance. A crossing in the opposite direction is also included, to specifically detail the impact of the relative mean wave directions and the relative wind direction. The third and final case is a simulation of the case presented by Bakke and Tenfjord (2017) for the original model. A comparison to the results of the original model is included to see if the model behaves in a similar manner, and thus give an estimate as to the validity of the simulation model.

All simulations were performed with the vessel presented for the original model, an open hatch general cargo vessel named Start Lysefjord(Bakke and Tenfjord, 2017; Grieg Star, 2013). The results of the different cases are presented in the following sections.

6.2.1 Case I: Seasonal weather variations

Case I is a set of simulations performed with varying start date of one-month intervals throughout 2017, with the configurations as shown below. The simulations were performed on a route between Aalesund, Norway and Reykjavik, Iceland. A route created by the RouteCreator script, with manual input of the port coordinates. A map of the route can be seen in figure 6.9.

Start date:	1. of each month through 2017, 06:00
Time step:	1 hour
Set speed:	14 knots

Figure 6.10 shows the average and maximum significant wave height encountered on each trip, and the significant wave heights encountered during four trips, each with a start

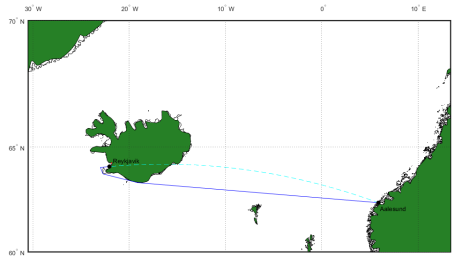


Figure 6.9: Case I - Route between Aalesund, Norway and Reykjavik, Iceland

date separated by three months. In figure 6.10a, both the average and maximum wave height is greatest in January and smallest in July, which in these latitudes equates to mid-winter and mid-summer respectively. The general trend is smaller average wave height during the spring and summer months, and greater average wave heights from October and through the winter months. Furthermore, the variations are greater during the winter months, with the extreme value in January being over three times the extreme value in July. The same trend is also reflected in figure 6.10b, where the variations in wave heights are clearly greater during the January and October trip, while being lowest for the July trip.

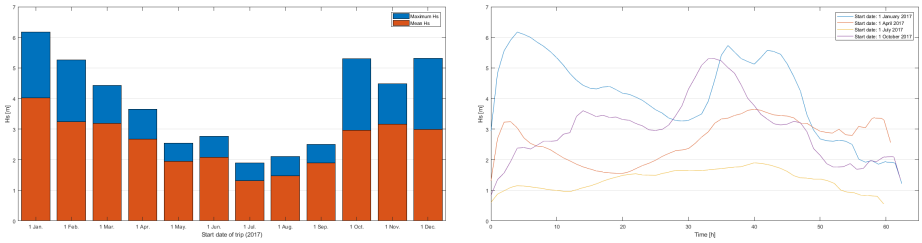
Similar plots for the resultant wind speeds are presented in figure 6.11. Both the average and maximum wind speeds are lowest in July and greatest in January, as shown figure 6.11a. The trends thus reflect that shown for the significant wave heights. The difference between the average and the extreme value is also grater for the months October through February, and also for April. Figure 6.11b shows the resultant wind speeds encountered during the same four trips as before. Not surprisingly, the experienced wind speed generally fluctuates more than the encountered wave height for any trip. Of the four trips included, the July trip experience the clearly lowest variation, and a maximum wind speed of about nine metres per second. The other trips experience great variations in wind speeds, with maximum values around 15-18 m/s, and minimum values around 2-3 m/s.

Figure 6.12a show the average and minimum values of the attainable speeds for each trip, while figure 6.12b displays the same data as average and maximum speed loss due to waves. The average attainable speeds show slight variations, with a maximum of about 13.8 knots for the September trip, and a minimum of about 12.9 knots for the March voyage. The greatest speed loss occurred during the trip in February and March, while the lowest speed loss was experienced in September. The blue bars in figure 6.12b show that the greatest maximum speed loss was experienced for the trips during October through April, with the exception of the trip in December. The latter trip experienced average and maximum speed loss on par with that of the July and August trips. There is no obvious correlations between the average wave height, and wind speeds shown in figures 6.10 and 6.11, and the average speed loss in figure 6.12b.

A plot showing the percentage distributions of relative wave directions, combined with the average significant wave height and average speed loss experienced during each trip, is displayed with figure 6.14. The wave directions are divided in the four categories used in the algorithm for speed loss estimation. The February voyage experience the grates amount of head sea, with about 16.5 % of the time, while the trips in April and September

experienced no head sea. The amount of following sea experienced was clearly greatest during the December voyage, at almost 70 % of the time.

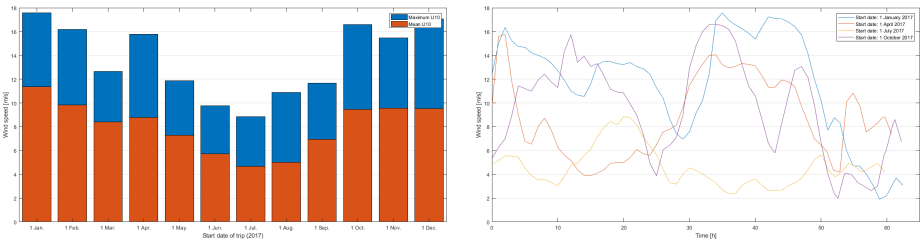
Figure 6.14 illustrates the combined effects wave height and wave direction on the average speed loss due to waves. The trips in October, November and December experienced similar average wave heights of about three meters. Comparing the trips during October and November shows that the greater amount of head and bow sea experienced during the October trip correlates with a greater average speed loss. Also, the December voyage experienced a lower average speed loss experienced compared to the November trip, while encountering an equal amount of head waves. The difference in speed loss correlates with a substantial difference in the amount of following sea experienced, which is about twice the amount in December compared to the trip in November. Comparing the months January and February, also shows that almost one metre lower average wave height for the February trip is offset by the more favourable wave directions during the January trip, resulting in a slightly lower speed-loss for the latter.



(a) Average and maximum wave height for each trip

(b) Wave heights encountered during four different trip

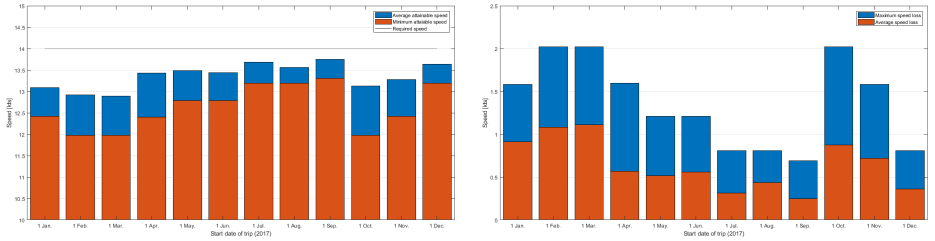
Figure 6.10: Case I - Significant wave heights encountered



(a) Average and maximum wind speed for each trip

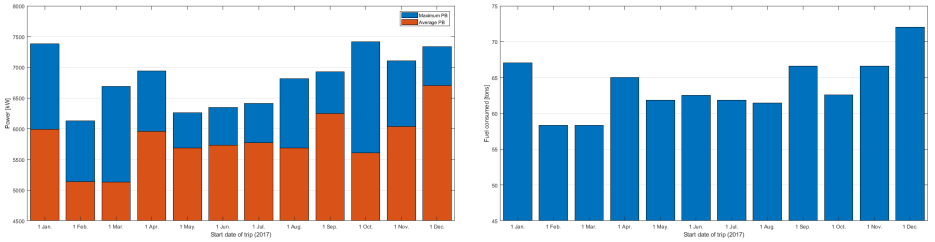
(b) Wind speeds encountered during four different trip

Figure 6.11: Case I - Resultant wind speeds encountered



(a) Average and minimum attainable speed **(b)** Average and maximum speed loss

Figure 6.12: Case I - Attainable speed and speed loss for each trip



(a) Average and maximum brake power **(b)** Fuel consumption

Figure 6.13: Case I - Brake power and fuel consumption for each trip

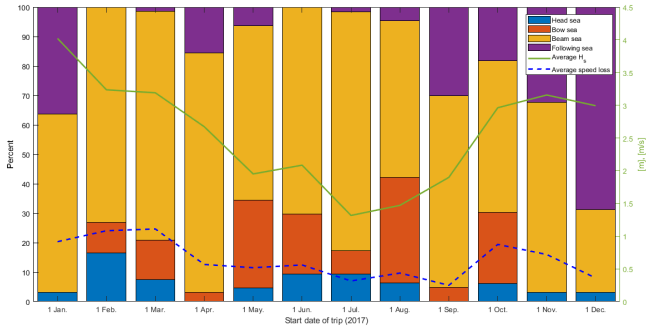


Figure 6.14: Distribution of relative wave directions compared to average H_s and average speed loss due to waves, for 12 trips during one year.

A juxtaposition of the average and maximum brake power, and the total fuel consumption for each trip, is presented in figure 6.13. There is naturally a correlation between the average brake power and the fuel consumption, as the latter is a function of the former. There are also similar trends between the average attainable speed and the average brake power, but not an absolute correlation. There is no clear correlation between either the

average and maximum brake power and the wind speeds in figure 6.11a, or with the wave heights in figure 6.10a.

6.2.2 Case II: Rotterdam - New York

The second case study is a simulated crossing of the Atlantic ocean, with a route from Rotterdam to New York, as shown in figure 6.15. The route was created by a previous iteration of the RouteCreator script, with ports chosen from the list of pre-defined ports, and the maximum length of the sailing legs set to 700 nm. Several simulations was performed in both directions for the same route, and with equal start date. Loading met-ocean data and running the simulation took on average 5 and 32 seconds, respectively. The simulation configurations used are shown below.

Start date:	01.01.16, 06:00
Time step:	1 hour
Set speed:	14.5 knots

Figure 6.16 displays the attainable speed, the significant wave height and the mean wave direction, encountered during both trips. The upper plots show the attainable speed varying in a distinctly discrete manner, and is not changing with each event, or simulation update. Rather being constant for up to approximately 24 hours. The lower plots show the mean wave direction relative to the vessel heading. The direction is given in absolute values such that it equates the angle of incoming waves on either side of the centre line.

From the middle plot in figure 6.16a it can be seen that significant wave heights of up to ten metres are encountered at one point during the voyage. The attainable speed is not visibly affected by such wave heights, when sailing in following sea. The return trip encounters manly head seas as seen in the lower plot of figure 6.16b. In this case there is a clear reduction in the attainable speed when the wave heights increase above a certain threshold.

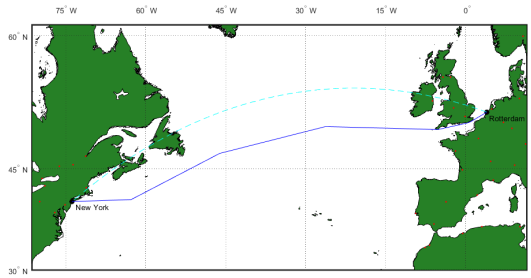
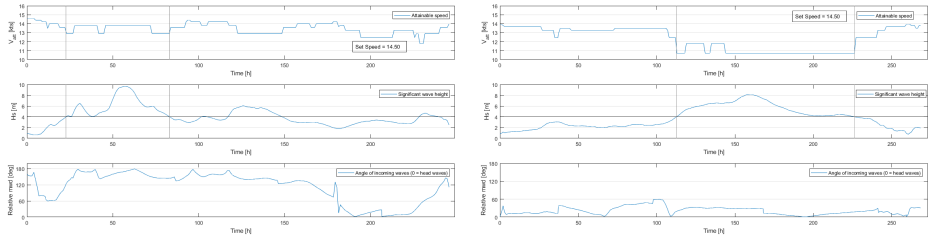
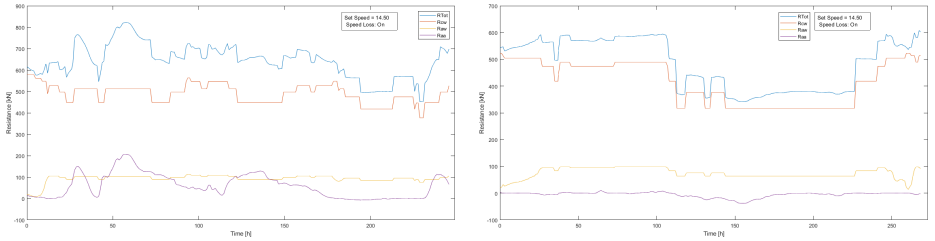


Figure 6.15: Case II - Route between Rotterdam and New York



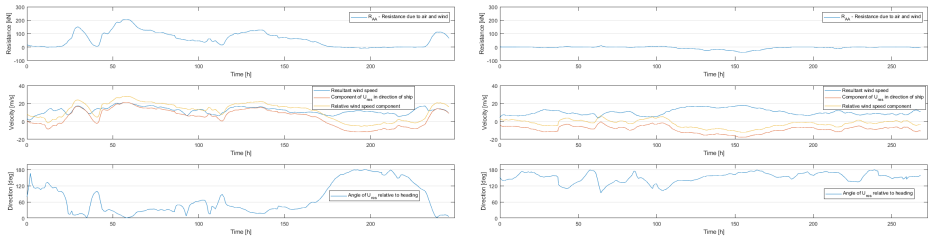
(a) Rotterdam - New York **(b)** New York - Rotterdam

Figure 6.16: Case II - Attainable speed, significant wave height and mean wave direction



(a) Rotterdam - New York **(b)** New York - Rotterdam

Figure 6.17: Case II - Total resistance and resistance components



(a) Rotterdam - New York **(b)** New York - Rotterdam

Figure 6.18: Case II - Air and wind resistance, wind velocities and relative wind direction

A significant wave height above four metres is experienced for significant parts of each trip, as indicated by the constant line in the middle plot of each figure. Four metres wave height is the most likely encountered wave height at Beaufort number seven, which is the maximum Beaufort number at which Kwon's speed loss formula is valid.

Figure 6.17 displays the total resistance and the resistance components for each time step of the two simulated journeys. The purple line shows the resistance due to air and wind; the yellow line shows the added resistance due to waves. The calm water resistance

is displayed by the red line, and the total resistance by the blue line. The resistance due to air and wind is the greatest contributor to the variations in the total resistance, on the trip from Rotterdam to New York. While the calm water resistance accounts for most of the variations during the return trip. This is illustrated in figures 6.17a and 6.17b respectively.

Figure 6.17a also show the added resistance in waves appearing approximately constant, around 100 kN from around ten hours onward, during the trip between Rotterdam and New York. On the return trip the added resistance shows greater variability, while remaining constant during large portions of the trip.

The impact of wind speeds and wind direction, on the resistance due to air and wind, is illustrated in figure 6.18. The upper plot displays the air and wind resistance during the journeys. In the middle plot the resultant wind speeds are shown. The total wind speed is displayed by the blue line, while the wind speed component in the direction of the ship heading is shown by the red line. Positive values indicate head wind, and tail wind have negative values. Combining the wind speed and the attainable speed of the vessel yields the relative wind speed in the direction of the ship heading, shown by the yellow line in each middle plot. The relative wind directions experienced during each trip is illustrated in the lower plots of figure 6.18. The direction is given relative to the ship heading in absolute values, such that head wind equates zero degrees, and beam wind on either side is given as 90 degrees.

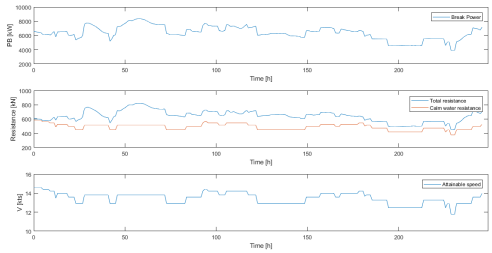


Figure 6.19: Case II - brake power, total and calm water resistance, and attainable speed during trip between Rotterdam - New York

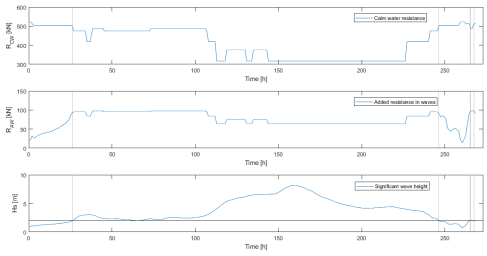


Figure 6.20: Case II - Calm water resistance, added resistance and significant wave height during trip between New York and Rotterdam

The trip from New York to Rotterdam experienced mostly tail wind, as seen in figure 6.18b. The result is little air and wind resistance and even negative resistance for some parts of the journey. The relative wind direction varies more during the trip from Rotterdam to New York. Figure 6.18a shows great increases in the air and wind resistance when the relative wind direction turns towards mostly head winds. It should be noted both trips encounter wind speeds up to around 20 metres per second. Conditions equating to a fresh gale and a number eight on the Beaufort scale.

Figure 6.19 displays the brake power, total resistance, calm water resistance and the attainable speed during the voyage between Rotterdam and New York. Not surprisingly, it can be verified that the calm water resistance is a function of the attainable speed, and that the brake power is a function of the total resistance.

Lastly, figure 6.20 displays a juxtaposition of calm water resistance, added resistance in waves and the significant wave heights. The results are for the trip from Rotterdam to New York. For wave heights above two metres the added resistance in waves is estimated as 20 percent of the calm water resistance. This is clearly illustrated in the figure, as the added resistance correlate with the calm water resistance for most of the trip. There is not a clear relationship between the significant wave heights and the added resistance in waves. The added resistance is lower for wave heights around eight meters than for wave heights around two meter, even at points where the relative wave direction is approximately the same, for instance at around the 75 hour timestep.

6.2.3 Case III: Comparison to original model

The final case study is a simulation of the voyage presented as a case for the original simulation model. The route between Yancheng, China and Panama, as seen in figure 6.21, is identical to that used by Bakke and Tenfjord (2017). The average time for loading met-ocean data and running the simulation with the new model was 2 minutes and 14 seconds and 6 minutes and 23 seconds, respectively. The time used to load met-ocean data varied greatly, with a median value of 25 seconds. The original model used on average 1 minute and 59 seconds for setting up the probability matrices used in the met-ocean data modelling, and 15 seconds for running the simulation. A comparison of the results from the two models is presented in the following.

Table 6.5: Performance and weather parameters for simulated journey with the old and new model.

Parameter	**Original model**	**New model**
Duration [h]	598	644
Average P_B [kW]	6096.5	5493
Max P_B [kW]	8257.5	7955.2
Fuel consumption [t]	645.3	626.3
Average speed [kts]	13.9	13.2
Average H_S [m]	2	2.8
Max H_S [m]	5.7	8.7
Average U_{10} [m/s]	6.1	7.9
Max U_{10} [m/s]	17	20.2

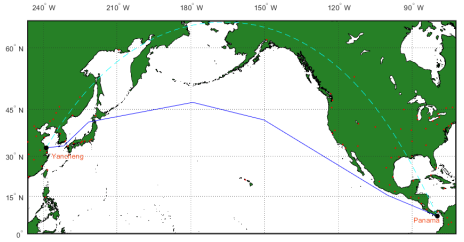


Figure 6.21: Case III - Route between Yancheng and Panama

The original model does not incorporate time, besides having a fixed time step between simulation updates. The matrices used to determine the weather parameters, are however based on met-ocean data for a given time period. For the original case study this time period is December 2016 and January 2017. The configurations used for the current simulation model is listed below.

Start date:	15.12.16, 06:00
Time step:	1 hour
Set speed:	14.5 knots

A comparison of the results from the two simulation models is presented in table 6.5. The simulated journey with the new model experience both higher average wave height and maximum encountered wave heights than the simulation with the original model. The same is true for average and maximum wind speeds. The simulation with the new model has a lower average speed of 13.2 knots, compared to 13.7 knots for the original model, leading to an increase in voyage duration of 46 hours. The simulation with new model also results in lower average and maximum brake power, as well as lower fuel consumption, compared to the results of the original model.

Figure 6.22 displays the distribution of the significant wave heights encountered during the simulated journey, with each model. The figure confirms that the new model experience both a wider range of wave heights as well as a greater average wave height.

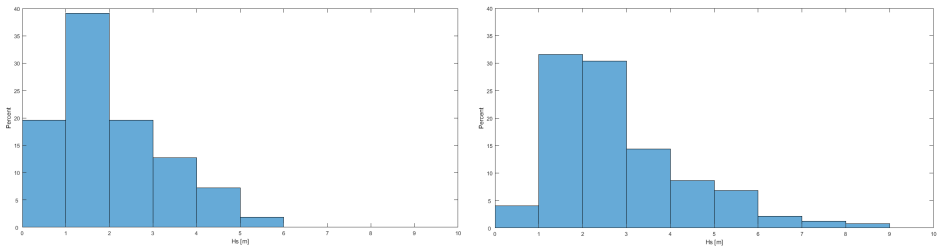
The distribution of the wind speeds experienced for each simulation is presented in figure 6.27. Almost half of the wind speeds experienced with the original model is between 4-5 m/s and between 6-7 m/s. While the wind speeds experienced with new model is more evenly distributed and takes a wider range of values.

Figure 6.24 presents a comparison of the attainable speed, significant wave heights and mean wave directions, experienced during the two simulations. The wave direction is again given in absolute values relative to the ship heading. From the middle plot the significant wave height varies more frequently and more abruptly for the original model than for the new model. The same holds true for the relative wave direction, with some exceptions. In the lower part of figure 6.24, abrupt changes in the relative wave direction experienced with the new model, can be seen at around 400 hours and also towards the end of the trip. The attainable speeds for both models seem to vary in a similar and discrete manner. Greater wave heights in mostly head seas seems to correspond with greater speed reductions. The simulation with the new model experience both a greater

span of significant wave heights and a greater variation in the attainable speeds, compared to the simulation with the old model. The wave heights experienced with the new model are above four meters for significant parts of the journey which corresponds to the periods of the lowest attainable speeds.

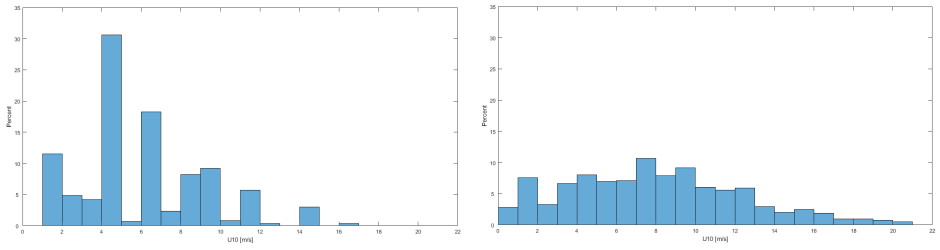
A comparison of the attainable speed and the calm water resistance, resulting from both models, is presented in figure 6.25. The calm water resistance is a function of the speed, and the figure confirms that the parameters correlate in a similar manner for both models. The calm water resistance for the new model shows a greater overall variation, compared to the new model. Figure 6.25 illustrates this as a result of a greater overall variation in attainable speed for the new model compared as compared to the original model.

A juxtaposition of the calculated resistance for the simulations with each model is presented in figure 6.26, with the resistance components displayed in similar colours as before. Reflecting the stochastic modelling of the weather parameters, all resistance components for the original model, seen in figure 6.26a varies more frequently than those for the new model in figure 6.26b. A greater variation in the overall resistance can be seen in the latter, correlating mainly with a greater variation in the calm water resistance for the new model, as presented in figure 6.25. The added resistance in waves is calculated as a fraction of the calm water resistance for most of the journey simulated by the new model, and also for substantial parts of the simulation with the old model. This can be seen by comparing the yellow and red lines in figure 6.26.



(a) Original model **(b)** New model

Figure 6.22: Case III - Significant wave heights encountered during voyage with old and new model.



(a) Original model **(b)** New model

Figure 6.23: Case III - Wind speeds encountered during voyage with old and new model.

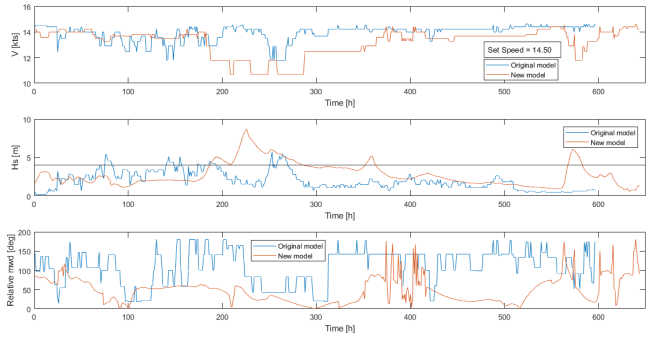


Figure 6.24: Case III - Attainable speed, significant wave height and mean wave direction during simulated journey.

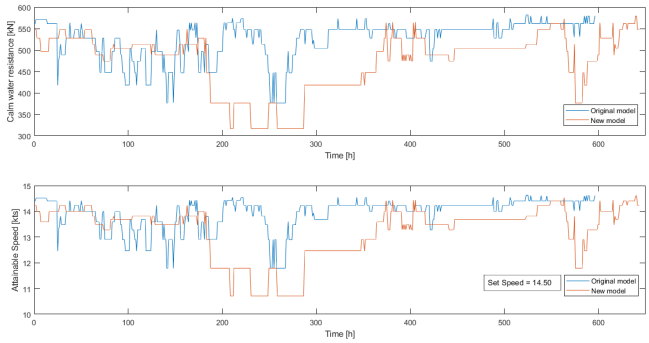
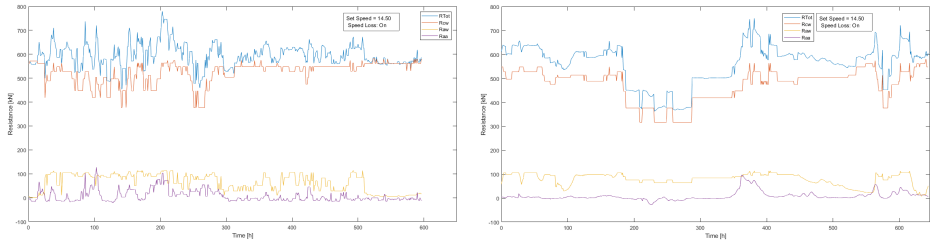


Figure 6.25: Case III - Calm water resistance and attainable speed during simulated journey.



(a) Original model **(b)** New model

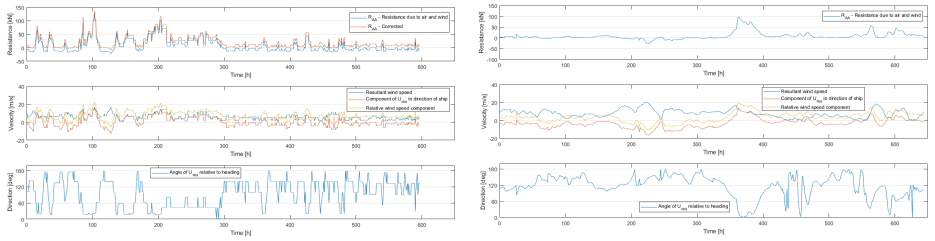
Figure 6.26: Case III - Resistance during voyage with old and new model.

Besides the differences in the incorporation of met-ocean data, there is only one difference in how the two models calculates the performance parameters. Namely in the calculation of resistance due to air and wind, as explained in section 5.2.4. Figure 6.27 illustrates how the calculated air and wind resistance corresponds to the experienced wind

speed and relative wind direction. The stochastic nature of the met-ocean parameter modelling for the original model, can again be seen in figure 6.27a. Resulting in more frequent variations in both resultant wind speeds and wind directions as compared to the results with the new model, seen in figure 6.27b. The lower plot of figure 6.27a illustrates how the relative wind direction fluctuates quite a lot between head wind and tail wind, during the simulation with the original model.

The upper plot of each figure shows the calculated resistance due to air and wind, during the two simulations. The air and wind resistance calculated with the original model is only the difference in air resistance due to wind, not including the air resistance due to the vessel velocity. As a result the calculated air and wind resistance is negative for all parts of the voyage where the wind speed component in the direction of the ship is negative, as illustrated by the blue line in the upper plot of figure 6.27a. The red line shows the wind and air resistance corrected to include the air resistance resulting from the vessel velocity.

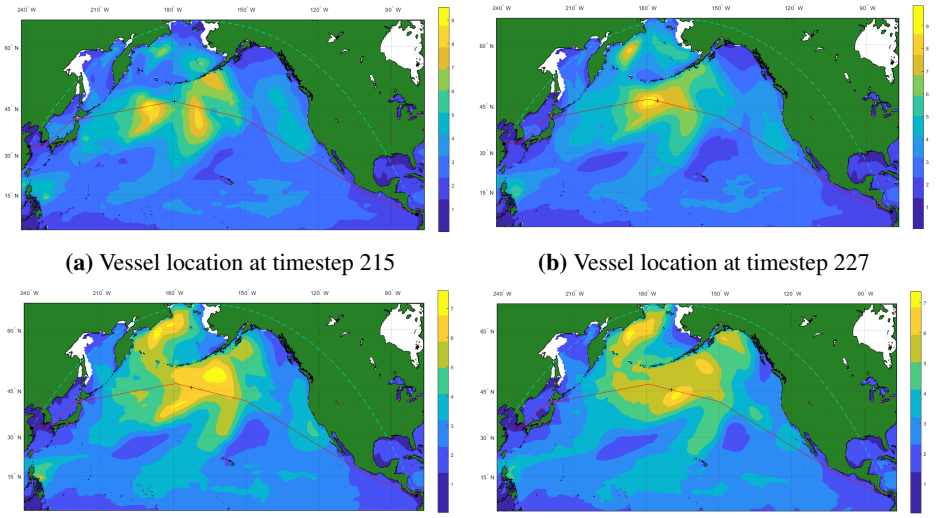
The simulated journeys with the original and the new model, experience maximum resultant wind speeds of 17 m/s and 20.2 m/s respectively. The maximum wind speed in the direction of the ship, relative to the ship position, is however greater for the simulation with the original model. With a maximum of 22.8 m/s for the original model, compared to 19.2 m/s with the new model. Resulting in a significantly greater maximum air and wind resistance for the simulation with the old model. The minimum experienced relative wind speed in direction of the vessel, is also lower for simulation with the new model. Corresponding to a lower minimum resistance due to air and wind, with the results from the new model compared to the original model.



(a) Original model **(b)** New model

Figure 6.27: Case III - Air resistance, wind speeds and wind direction during simulated voyage.

Figure 6.28 illustrates the significant wave heights present at the time of four locations along the simulated journey with the new model. Given a fixed route there is no way of avoiding rough conditions. In figure 6.28b the vessel is display at the timestep of maximum significant wave height, corresponding to the peak seen in the middle plot of figure 6.24.



(a) Vessel location at timestep 215

(b) Vessel location at timestep 227

(c) Vessel location at timestep 239

(d) Vessel location at timestep 251

Figure 6.28: Surface plots showing H_S at four timesteps approximately 12 hours apart.

Chapter 7

Discussion

This chapter is devoted to a discussion of the results presented in chapter 6.

7.1 Route-finding algorithm

Creating a universal algorithm for setting up routes between any two ports is an extensive problem, given the all considerations that must go into creating a feasible and near optimal route. When only looking at the shortest distance, the problem is still challenging. It is difficult to create a universal algorithm that incorporate all special cases that can arise when considering possible port locations relative to each other and to the surrounding land areas, the possible combinations of the shapes of intersected land areas, and the possible discretisation of the shoreline sailing path polygons. The presented algorithm and results should be evaluated in context of expected scope of the master thesis. The results indicate that the algorithm can be significantly improved by the means discussed in the following sections. Nevertheless, the main principals of the algorithm have been demonstrated, and the acceptable routes compared to routes created by an online tool showed a good match. Only a small sample of routes were however included in the comparison due to time constraints. A more comprehensive comparison would be valuable for evaluating the results.

7.1.1 Route fitness

When restricting access to polar waters, 62.6 % of the unique routes created between the included ports were deemed acceptable. Leaving a high number of 37.4 % of the routes as non-acceptable. 5.3 % of all routes had two issues making them non-acceptable. For the 27 routes through polar waters, between the included ports, 44 % were acceptable and 66 % non-acceptable. The routes with two errors made up 7.4 % of all routes.

The acceptability of the resulting routes is based on visual inspection and a subjective evaluation of the fitness of the route. Routes are deemed acceptable if corresponding to a near shortest path, while having no unnatural sharp turning-points. In real life, routes

are based on several other factors besides being the shortest path. Meteorological and oceanographic conditions, political factors, wars and pirate activity, must all be considered when determining the best route for a given vessel and cargo. Without special knowledge and experience it is difficult to make a judgement on if a specific route is feasible or acceptable, and certainly if it is optimal. Incorporating the mentioned considerations into a universal algorithm is also a non-trivial task, given all the special cases that must be accounted for. It would be possible to avoid generating routes through problematic areas, such as war zones and areas of pirate activity, by incorporating known issues when defining the shoreline sailing path polygons. Should the presented method be further developed, it would be of value to have persons experienced in setting up routes involved in defining the shoreline sailing path polygons. Incorporating political or cultural conditions in the definitions would however trigger a need for regular updates as conditions change.

Given the dynamic nature of the meteorological and oceanographic conditions, the optimal route for a given vessel will vary with changing conditions. Tanaka and Kobayashi (2017) highlights the fact that the optimal route in the presents of ocean currents can deviate from the great circle route. The optimal route will thus change with seasonal changes of ocean currents and will also change depending on the direction when sailing between two ports. When incorporating the effect of wind, waves and currents, the process becomes a weather routing problem. Finding an optimal route for a given vessel in specific conditions is a different problem than evaluating vessel design performance on a given route with the given conditions. Although an interesting problem, it is not within the scope of this thesis.

Another aspect of evaluating routes, is that a route may be acceptable for one vessel while being non-feasible for another vessel. If for instance the draught of vessel is too great for a canal or the machinery prohibits a vessel from sailing through emission control areas (ECAs). Customised routes could be achieved by adding vessel dimensions as a limiting factor for canals to the algorithm, and by including an option for allowing sailing through ECAs. Another non-trivial task, but a possible solution is to add alternative sailing path polygons in the case of restrictions, in a similar manner as it is done for the Antarctic continent when restricting sailing in polar waters. For instance if vessel dimensions prohibit sailing through the Panama Canal, the sailing path polygons around the Americas could be merge to a single polygon to force a route around South America.

Fitness of a route also depends on the objective for creating the routes. It may not be of great importance that a route has unnatural sharp turning point near a port when simulating vessel performance. If the route is used to compare the performance of two different designs, small deviations from an optimal route are unlikely to have great impact on the comparison, when using an identical route. It can be argued that it is mostly important to get the offshore part of a route correct when evaluating the impact of environmental conditions on the vessel performance since the impact of wind and waves generally are greater further from shore.

7.1.2 Erroneous results as related to the algorithms

Most non-optimal routes are caused by two common types of errors. Type-1 errors are most prevalent and occur when sub-optimal connection nodes are chosen for connecting a port to a shoreline sailing path polygon. Type-2 errors occurs from a sub-optimal choice of

which land area to first circumvent, resulting in a route of greater distance than necessary. There are additional errors that repeatedly produce non-optimal routes. It is logical to suspect that repeating errors are produced as a consequence of the applied algorithm. A discussion of different repeating errors in relation to the algorithms used, is presented below.

Sub-optimal connection nodes

Connection nodes are determined from a set of different algorithms. A specific algorithm is chosen based on the locations of the ports relative to each other, and whether the ports are visible to each other. All the algorithms choose connection nodes from the nodes of the shoreline sailing path polygons. Erroneous routes from sub-optimal connection nodes can therefore both be a result of the chosen algorithms for establishing connection nodes, as well as poorly defined shoreline sailing path polygons.

The location of nodes in the sailing path polygon in relation to the port location has great influence when generating a path omitting the land area where the port is located. A poor match will yield sub-optimal paths at the start or end of a route. If the sailing path polygons are not properly defined for an intersected land area, it is likely that the connection node found are sub-optimal, irrespective of the method for establishing the connection nodes. It is a challenge to define the sailing path polygons in such a manner it is possible to establish suitable connection nodes for routes between an origin ports inside the polygon towards all possible destination ports.

When the ports are removed from the land area and not visible to each other, the convex hull approach is an efficient method to reduce the possible number of paths around the land area. The method is however only exact for nodes on a plane and there can be instances where using the convex hull approach results in sub-optimal connection nodes. If this occur, it is likely that an adjacent node is the optimal fit, and the problem is thus not likely to result in great differences in the overall distance. For all other cases, the algorithms for determining connection nodes are less precise. The heuristics used are intended to ensure near shortest paths, suitable in most instances. Poor choices will however occur, and few of possible issues are discussed in the following.

When a port lies inside the sailing path polygon while not visible to the other port, the number of nodes evaluated as candidates for connection nodes might not be sufficient. The nearest nodes plus one, two or three adjacent nodes in each direction are chosen based on the number of nodes in the sailing path polygon. Unless both ports are located inside the ailing path polygon and have candidate node in common, the furthest adjacent nodes in each direction is chosen as the connection nodes. There are cases when an even further adjacent node would be a better fit for connection nodes. This is illustrated in figure 7.1, where the node marked by the red cross would be the optimal choice for a path in the clockwise direction from the inside port. In most such instances the route direction from the port will be approximately correct, while not being optimal. A more prevalent source of routes with unnatural turning points are when the location of nodes of the sailing path polygon is a poor fit for the port location. Figure 6.3a illustrates this issue for routes from Valencia through the Strait of Gibraltar. The same problem can also occur when both ports lie inside the sailing path polygon and share a candidate node in common.

Another issue occurs when both ports lie inside a sailing path polygon around a land

area intersected by a great circle route between the ports, while the ports are visible to each other by a rhumb line. In such cases it is not logical that the route should go by the shoreline sailing path polygon if a rhumb line route is shorter. The method is chosen to comply with the overall algorithm. It is not known if the described case occurs often. The ports will likely be located in close proximity to each other, and the difference in distance may thus be negligible.

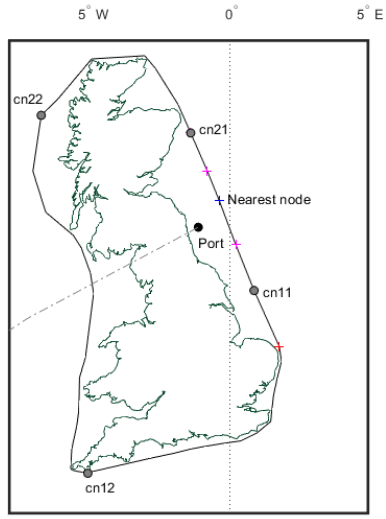


Figure 7.1: Non-optimal choice of connection node for port inside sailing path polygon, when ports are not visible to each other.

Non-optimal order in which to circumvent land areas

Type-2 errors usually occurs when the route intersects several land areas, and a poor choice is made for which land area to first circumvent. The chance of substantial type-2 errors increases if the initial route intersects one or more continents, since omitting a greater land area implies increased likelihood for a greater deviation from the shortest path, compared to when omitting a smaller land area. A clear example is the route between Durban and Rotterdam shown in figure 6.3c, where the African continent is chosen as the land area to first circumvent, then the European continent. The resulting route thus goes thought the Suez Canal and the Strait of Gibraltar, which is a longer, more time consuming and costly route. Deviations in the shortest path route can also occur in later iterations, since the connection nodes for a shortest path between two land areas might not be an optimal solution when circumventing an intersected land area between the two first land areas.

The vulnerability for type-2 errors is clearly a result of the chosen methodology of circumventing one land area at the time in an iterative manner, as compared to a holistic approach. The algorithm for selecting which land area to circumvent is based on heuristics, partly developed by trial and error, with the objective to maximise the chance of a good selection. Type-2 errors made up about 14 % of the errors for results presented for routes

between the included ports when restricting access to polar waters. A possible solution to the problem of type-2 errors could be creating several routes, varying the order of the land areas to be circumvented, and then selecting the shortest route. This solution will however increase the processing time, since the route-finding process will be multiplied. An alternative is making the problem negligible by only altering the order of the first and second land area to omit, to limit the increased processing time.

Shoreline sailing path polygons

The ability of the sailing path polygons to model a suitable routes along the coastlines is essential to the algorithm. Good results depend on the definitions of the sailing path polygons being a good fit for the specific problem. This is especially important if a port is located inside a sailing path polygon. A challenge with using predefined sailing path polygons, is that it is not possible to define a sailing path polygon in a manner that ensures suitable routes in all directions for a port located inside the sailing path polygon. The routes from Valencia through the Strait of Gibraltar exemplify the problem. The shorelines sailing path around the Eurasian continent cannot be defined, in the Mediterranean ocean, in a way that ensures both suitable routes from the Gulf of Valencia through the Strait of Gibraltar, and suitable routes from Gibraltar to the Suez Canal. This is a fundamental challenge of the chosen methodology. The issue of defining universally suitable sailing path polygons is especially challenging for the large continents, which is where most large ports are located. Again, a possible solution is to define several sailing path polygons, and select a suitable polygon based on the port locations. With the current method of using one shoreline sailing path polygon per land area, the best one can do is perhaps to optimise polygons for routes between the largest ports on each continent.

Besides errors resulting from a poor fit of port locations and the definitions of sailing path polygons, additional errors can occur based on the shoreline sailing path polygons. Only 28 of the sailing path polygons have been modified from the initial convex hulls made around each shoreline polygon. Most of the sailing path polygons is thus still defined as convex hulls, which means that they are not inspected as feasible sailing paths. Overlapping convex hulls are a potential risk of errors, since a route along one land area then is likely to intersect another land area. Since most major land areas has modified sailing path polygons, the mentioned errors are not likely to cause a great deviation in the overall distance, compared to a feasible route omitting the intersected island. The precision level of the sailing path polygons beyond a certain point, is not crucial for demonstrating principle behind the route generation algorithm, or for presenting valid results. Still it is obvious of great importance to avoid all land areas when defining shipping routes. Ideally all convex hulls with more than a few nodes should therefore be modified to model feasible sailing paths.

Shoreline polygons

The shoreline polygons used in the route-finding algorithm is based on GSHHG data of intermediate resolution. The intermediate resolution polygons have far fewer nodes than the polygons from the full resolution dataset, and polygons used are therefore not precise. The resolution of the intermediate data is arguably high enough to avoid great errors when

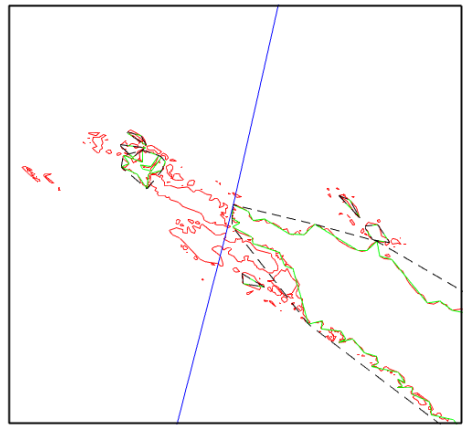


Figure 7.2: Unfeasible route due to missing shoreline polygons, an unmodified convex hulls as sailing path polygons.

circumventing land areas, since higher resolution will not greatly alter the outline of the land areas. Furthermore, the modified shoreline sailing path polygons have been generated to comply with the full resolution shoreline data.

Of greater concern is the number of polygons included in the dataset used in the route-finding algorithm. Only the 5701 level one polygons from the low-resolution data is included, while the full resolution data includes 179832 level one polygons. A tremendous number of islands are missing, mostly small island near shore of other lager land areas. Problems can arise even if the sailing path polygons comply with the full resolution map data, by small islands missing when checking for the route for intersections. The route between Coln and New York, presented in chapter 6, is an example such error. Another example of missing shoreline polygons leading to error is included the presented results, on a route between Sydney to Felixstowe via the Northeast Passage. Figure 7.2 illustrates the issue, where the red shoreline polygons are from the high resolution data set and the green shoreline polygons are those used in the route-finding algorithm. Also, the convex hulls around the included land areas has not been modified to ensure feasible sailing paths.

Both the resolution of each shoreline polygons and the number of polygons to include is a question of necessary route precision versus computational demand and time consumption. There is a clear trade-off between precision of the results and computational demand.

Other errors

Port paths are included in the algorithm to allow for route between ports at locations without access to the open seas. It is difficult to define the port paths to be a good fit for routes in any direction. As a result, the pre-defined port paths can be source of sub-optimal routes. An example is the route between Mumbai and Dubai illustrated in figure 7.3. A possible solution to the problem is to let the user define custom paths for each port. This would have the additional advantage to allow for a wider range of customised port locations.

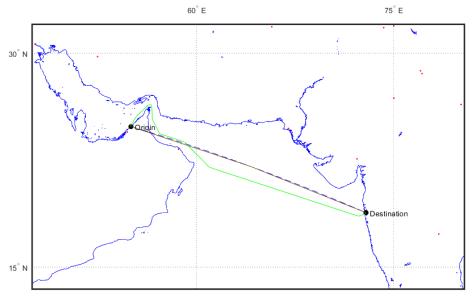


Figure 7.3: Non-optimal route due to the pre-defined port paths.

The route presented between Dubai and Los Angeles reveals an algorithmic weakness that can produce non-optimal routes between ports located on nearly opposite longitudes of the globe. The direction of the great circle route between the ports, determines the east-west direction of the final route. In some cases, it is shorter to circumvent the globe in the opposite direction of the great circle route. This can occur for routes with great longitudinal span where a route in the same direction as the great circle route, involves circumventing several large land areas. Since the error seems to be rare, the best solution might be to modify the route by adding a suitable waypoint in the opposite direction of great circle route.

7.1.3 Modifying resulting routes

The purpose of the route-generating algorithm is to enable setting up routes for use in the simulation model. Having the algorithm produce non-satisfactory routes more than a third of the time is arguably not a great result. To amend this problem, the RouteCreator script includes an option to modify the initial resulting routes. All routes made non-acceptable by a single issue can be satisfactory modified by either altering the maximum leg length or defining a path of waypoints to be included in the route. The routes made non-acceptable by two issues may however not be amendable by the modification process. By looking at the results for the routes between the predefined ports, 5 to 8 % of created routes may be left as non-satisfactory by the chosen standard. Expanding the list of predefined ports and testing the script for all combinations, would yield a more precise number of the percent of non-amendable routes.

The ability to modify routes have been demonstrated by modifying six non-acceptable routes as presented in section 6.1.3. Five of the routes had single errors, while the route between Durban and Tokyo suffered from both a type-1 and type-2 error. Interestingly, both errors were fixed by adding waypoints forcing the route to go through the Sunda Strait. Double-error routes can thus be made acceptable in some cases by adding waypoints. Routes with two type-1 errors can never be properly amended with the current modification option. Adding the ability to modify routes by defining two paths of waypoints and forcing the final route to include both in proper order, should allow for amending most erroneous routes.

7.1.4 Methodology

The algorithm for automatic route-generation has been developed from solving a basic case of establishing a route around one land area, when the ports are removed from land and not visible to each other. The final method is the result of a stepwise expansion from the basic case to a more versatile model, circumventing several land areas and accounting for several different port locations. The work methodology is reflected in structure of the final overall algorithm, especially in the process of determining connection nodes for connecting the open sea parts of a route to the shoreline path. The process of choosing the order to circumvent the land areas, has also partly been developed in a trial and error approach.

The methodology for determining connection nodes is somewhat complex and consist of several different algorithms, all based on heuristics. A fundamental idea behind the presented method is to reduce the shortest path problem by limiting the possible number of arcs to generate by first finding suitable connection nodes. Using a convex hull around the ports and the shoreline sailing path polygon to identify connection nodes seems like a positive contribution to finding shortest path around a land area when both ports are removed from the land area and are not visible to each other. The connection nodes found from the convex hull approach seems to be suitable in most cases. For all other cases, the heuristics are less likely to yield optimal connection nodes, and the methods for establishing connection nodes are less intuitive, more complex and at some points somewhat arbitrary. The algorithm would benefit from a more generalised approach to finding connection nodes.

A holistic approach investigating all land areas in a region between ports, and all possible paths between the shoreline polygons, and between each port and all shoreline polygons, should in theory guarantee a shortest path solution. Besides being the result of the work methodology, the present heuristic approach is chosen with the intention of reducing the number of computations needed to find a suitable route. This is the justification both for first establishing connection nodes, and for circumventing one land area at the time. The problem of establishing a route by accounting for all possible land areas to omit is a far more complex problem than to circumvent one land area at the time. The number of computations needed for generating a route with each approach will vary from case to case, depending on the number of land areas and the number of nodes in the sailing path polygons. For the route between Colon and Singapore, 4667 sailing path polygons with a total of 32056 unique nodes were included in the dataset used to find the route. Creating a partial visibility graph and applying Dijkstra's algorithm would require a tremendous number of steps to compute. While the presented method inspected routes between 25 pairs of endpoints for intersections, only included 12 polygons to circumvent.

There is a trade-off between probability of optimal result and problem complexity. If this trad-off is justified, is a question of both the necessity for route accuracy and resource demand of solving the problem in a holistic manner with a shortest path algorithm. The level of route accuracy needed depends on the use case of the route. For use in a simulation model, it can be argued that the proposed algorithm with the modification option, is able to produce satisfactory routes in most cases.

7.2 Simulation model

The main motivation behind creating a new simulation model has been to ensure the ability of running simulations with routes created by the route-finding algorithm. The ability of the new model to create routes and simulate vessel performance along those routes is demonstrated by the results from case I and II. A discussion of the model behaviour is presented in the following, and then some thoughts on the model validity is given.

7.2.1 Met-ocean data modelling

To ensure the capability of simulating sailing along any given route divided into any number of sailing legs of different lengths, the need to pre-process the met-ocean data according to each route, which was the case for the original model, had to be eliminated. This is achieved by using hindcast met-ocean data directly.

The results from case III illustrates the differences in the two methods of modelling the met-ocean conditions. The distributions of wind speeds experienced during the simulations of the two models, as was presented in figure 6.22, shows a substantial difference. The more even distribution of wind speeds for the new model is likely a better reflection of a natural process. The wind speeds along each simulated journey, seen in the lower plots of figure 6.27, reflects the same notion. Where the wind speed during the simulation with the original model varies quite abruptly, compared to the more gradual development of the wind speeds during the simulation with the new model. The same holds true for the significant wave height and relative wave direction displayed in figure 6.24

Abrupt changes in weather parameter values, as seen for the original model, is to be expected from a stochastic approach. Even if the state of a Markov chain depends on the previous state, the state is ultimately decided by probabilities. Furthermore, there is a finite number of possible states a system can have. The met-ocean parameters are also assumed to be completely independent with the Markov chain approach used by the original model. This assumption is questionable. At least the wind generated waves should relate to the wind speed and wind direction. It is therefore the authors opinion that using hindcast met-ocean data directly reflects a more natural development of the weather conditions, and hence result in a more realistic model. Kauczynski (2012) argues in the same manner when he concludes that applying complete historical weather time series along a route, is the most reliable approach for studying the reliability of ship transportation.

7.2.2 Attainable speed

The attainable speeds obtained during all journeys presented from case II and III clearly variate in a discrete manner. This is a direct function of the formulae used for estimating speed loss due to waves. The algorithm yields discrete values for the estimated speed loss, as can be seen in figure 7.4, where the attainable speed is presented as a function of the significant wave height and mean wave direction. The values shown are calculated with the algorithm from the original model.

The results presented for case II in figure 6.16, shows relative small variations in the attainable speed for varying wave height, when the vessel sails in mostly following sea from Rotterdam to New York. In fact, wave heights up to ten metres in following sea has

no visible effect on the attainable speed. For the trip in the opposite direction, experiencing mostly head sea, there is a significant speed loss due to waves when the significant wave height goes above four metres. There is however no difference in the attainable speed in head waves of four and eight meters wave height, as seen in the middle plot of 6.16b between 150 hours and 200 hours. The reason for this is that the calculated attainable speed is partly based on the Beaufort Number, which in the algorithm is set equal to seven for all significant wave heights above four metres. The result is that for any given relative wave direction, the attainable speed is equal for all waves of significant wave height above four metres.

Bakke and Tenfjord (2017) states that the speed loss estimate will not be accurate for Beaufort numbers above six. According to the Beaufort scale, the probable wave height is four metres av Beaufort number seven(metoffice.gov.uk, 2020). Presumably this is the reason why the Beaufort number is set at seven for any significant wave height above the four-meter threshold. Though it is difficult to see how this would make the estimate more reliable. Results should therefore be viewed with caution for any simulation where the significant wave heights are above four meters for a significant part of the voyage.

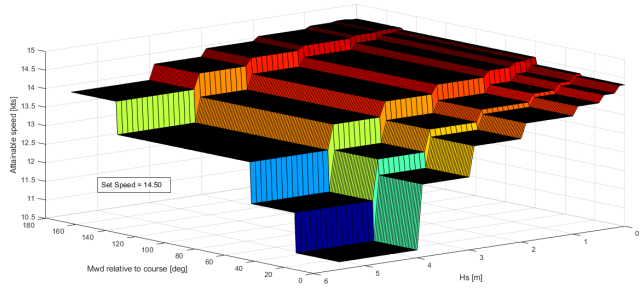


Figure 7.4: Attainable speed in different wave heights and relative wave directions

Effects of combined wave data

The significant wave heights and relative wave directions are for combined swell and wind waves in the present model. The original model loads wave data separately for swell and wind waves and uses combined values for the resistance calculations, calculating the attainable speed stored to the MATLAB workspace. The significant wave heights and relative wave directions used to calculate the attainable speed as a simulation variable are however for wind waves only. The Beaufort scale relates to well-developed wind waves (metoffice.gov.uk, 2020), and since the speed loss estimate depends on the Beaufort number, it may be appropriate to use wave heights for wind waves only when estimating the attainable speed. The effect of using wave data for combined wind and swell in the speed loss estimate should be further investigated. The simulation model would in any case be improved by allowing for wind generated waves and swell loaded separately, such that the ShipX module from the original model can be used.

7.2.3 Resistance calculations

At significant wave heights above two meters, the added resistance in waves is estimated as 20 % of the calm water resistance. Since the calm water resistance is a function of the speed, the total resistance and brake power is dominated by the estimated attainable speed. This is illustrated in figure 6.20. The model thus behaves such that increasing wave heights in head and bow seas result in reduced attainable speeds, where significant speed reductions occurs for wave height above about 2-3 meters. At such wave heights the added resistance in waves is taken as a fraction of the calm water resistance. The reduced speed thus leads to a reduction in both calm water resistance and added resistance due to waves and a reduction in brake power.

The methods for evaluating the ship performance are kept from the original simulation model, and it is beyond the scope of this thesis to evaluate the validity of the methods used for speed loss and resistance calculations given by Bakke and Tenfjord (2017). It will however be noted that it seems counter-intuitive to get reduced added resistance in waves with increasing wave heights. From the authors perspective it would make more sense to calculate the resistance with a fixed speed, where increased wave heights would yield an increased added resistance. A reduction in the attainable speed would only occur when the power needed to maintain the fixed speed surpass the available power. Alternatively, to have a fixed power output and calculate a variable attainable speed based on the resistance it is possible to overcome with the available power. A caveat must be added; resistance calculations has not been a topic of the thesis and the presented notions can therefore be based on an incorrect understanding of the subject.

7.2.4 Time consumption

Both the average time used to load the met-ocean data and to run a simulation varied significantly between the simulations in case II and III. Loading the met-ocean data to MATLAB and then to Simulink seems to dominate the time consumption. The length of the route determines the amount of met-ocean data loaded and is thus a determining factor for the time consumption. Met-ocean data is loaded for a rectangular area covering the extreme coordinates of the route. Since the route is fixed, this means that a great amount of unused met-ocean data is loaded. Reducing the amount of such data loaded to both MATLAB and Simulink, should be fruitful avenue to follow in the pursuit to reduce the time consumption. A possible solution is to load data for several smaller rectangular areas covering the route, as long as there is no need to pre-process the data before running the script executing a simulation.

The simulation model as it is now, with independent calculations of the attainable speed and resistance, could be simplified by performing the simulation only with the speed loss calculations. The performance parameters could then be calculated post simulation since these do not influence the location and date at each simulation update. With such an approach, the met-ocean data loaded to Simulink could be reduced to only include the significant wave height and the mean wave direction, likely to contribute to a lower time consumption. This would however make model less intuitive. Also, the goal of original model to have drag and drop functionality, to easily alter configurations, would be lost. The former solution is therefore rather recommended should the model be improved in the

future.

The average time used to load met-ocean data was similar for the original and the new model, while the median time used was about 1 minute and 30 seconds less with the new model. However it takes almost six minutes longer to run the simulation for the new model compared to the original. Since the main contributor to the difference likely is loading and processing hindcast met-ocean data in the Simulink model, it seems to be a trade-off between accuracy of the data and processing time. Even though six minutes can be a long time when running several simulations, it can be argued that the value added with a more realistic simulation model outweighs the downside of increased time consumption.

7.2.5 Validation

Validation of the simulation model will be discussed according to the principles by Sargent (2005), as presented in chapter 4.

Conceptual model validation

Conceptual model validation deals with the underlying theories and assumptions for the conceptual model, and if these are appropriate to represent the system. In this regard it seems poignant to emphasise that the method used for speed loss estimation is deemed inaccurate for Beaufort Numbers seven and above. Such wave height are experiences in all three cases presented in chapter 6. The method for estimating speed loss can therefore not be said to be valid over the domain of possible wave heights encountered. If the mentioned objections regarding calculating resistance and brake power based on a speed loss estimate are reasonable, the conceptual model should also be evaluated further in this regard.

Another issue regarding the conceptual model of speed loss is that voluntary speed reduction is not included in the model. It is safe to assume that voluntary speed reduction will take place in extreme wave height, depending on wave direction and vessel type. The conceptual model in this regard is thus not an accurate model of the system it represents. Besides reducing the speed in extreme conditions, it is reasonable to assume that a shipmaster in some cases will avoid extreme conditions when possible. In the simulation model the vessel will follow a fixed route independent of the weather conditions arising. An illustration of this can be seen with the surface plots presented in figure 6.28. Independent of the criteria used, the lack of ability to alter course along the journey in severe weather conditions does not accurately represent the real-world system.

There are several additional ways the conceptual model is an unrealistic representation of a ship sailing between ports. Waiting for time slots in canals and reduced speed in canals and ECAs are examples of aspects not included in the conceptual model. The conceptual model validation must however be evaluated considering the intended purpose of the model. And it can be argued that the latter aspects are of little significance for the purpose of evaluating the performance of a ship design at sea.

Computerised model verification

Walk-troughs of the MATLAB code and Simulink model have been performed to ensure correct functioning of the simulation model. The models ability to extract correct met-

ocean data for a given location and data have specifically been tested and verified. The simulation model executes without errors and produces the desired output variables. Based on testing, the model logic can be assumed to be in order, even if minor errors can have been missed.

System data validation

The model mainly relies on three types of system data. The route data is generated by the route-finding algorithm, while the vessel data used in all three cases is kept from the original model, and the met-ocean data is obtained from ECMWF. Besides the previously discussed issues with the validity of the route data, there is no reason to question the validity of the system data. A validation of the vessel data and the met-ocean data has therefore not been attempted.

Operational validation

The output behaviour of the model should ideally be compared to that of the system it represents. The accuracy of the output data is hard to evaluate without the operational behaviour system data, which in this case is lacking. The judgement of the operational validity must therefore be done based on the simulations performed for the three cases presented in chapter 6, discussed above. The results presented for case I illustrates that the encountered wave heights and wind speeds varies through the season in an expected manner, which implies that the model reflects the seasonal variations in the met-ocean conditions in an accurate way.

The comparison with the original model is limited by the fact that results are only available for one route for the original model. Also, the results presented for case I are based on one simulated journey per month, while the results for case two are based on simulation of one round-trip between two ports, and for case III one trip in one direction only. Running simulation repeatedly for longer time periods could yield more reliable information regarding the model output behaviour. This especially important should the model be used to evaluate a vessel design, since one design can perform better in the specific conditions present on one trip, while another can be better overall through a year. The results from case III does imply that output behaviour of the current model works in accordance with the original model. The vessel performance parameters, except for the air and wind resistance, responds in a similar way in similar met-ocean conditions. It is an expected result, since the methods for calculating attainable speed, calm water resistance and added resistance, are identical. The original model was evaluated against system data, with some deviating results. Since no such data has been made available for this thesis, the current model cannot be said to have not been validated with a high degree of confidence. Should the model be further developed, it would therefore be essential to validate the model behaviour against operational behaviour system data.

Chapter 8

Conclusions and Recommendations for Further Work

A new heuristic algorithm for generating routes between two ports has been presented. The ability to generate routes between a wide selection of port combinations has been demonstrated. Routes are produced in a short time in limited number of computational steps, compared to traditional shortest path algorithms which requires first constructing partial visibility graphs. The drawback of a heuristic approach is that optimal results are not guaranteed. Optimal in this regard, is feasible approximate shortest path routes. The results show that about 37 % of the resulting routes between the 20 included ports were non-acceptable, when restricting access to polar waters. About 86 % of the non-optimal had one error making them non-optimal, which means they can be redeemed by the option to modify the resulting routes. The majority of the non-optimal routes are a result of non-optimal connection nodes, and improving the algorithms for selecting connection nodes should improve the overall performance. A comparison to routes generated with an online tool and a selection of acceptable resulting routes, showed a good match.

The route-finding algorithm has been implemented in a simulation model for evaluating vessel performance in realistic conditions, and the ability to generate routes when running the simulation model has been demonstrated. Results from the three case studies have been presented. The results show that the output performance of the new simulation model is comparable to the original model for similar met-ocean conditions, with the exception of the air and wind resistance which has been corrected in the new model. Applying hindcast met-ocean data directly resulted in a more realistic modelling of the met-ocean conditions in the new model, when compared to the original. A drawback of applying hindcast met-ocean data is a significantly increased computational time demand, depending on the distance of the route used in the simulation. The method for estimating speed loss due to waves, does not seem to reflect variations in wave heights in waves of significant wave heights above four metres. The results for simulated journeys in such condition can therefore not be said to be reliable. Finally, the model has not been validated against system data.

8.1 Recommendations for Further Work

Several avenues for improving the route-finding algorithm has previously been discussed. Recommendations for improvement is listed below.

- Suggestions to improve the functionality and results:
  - Adding functionality to let the user define port paths in a desired direction to minimise problem of poorly defined connection nodes, and to avoid a poor fit between route direction and the predefined port paths.
  - Adding the capability to modify the generated routes by adding two paths of waypoints to ensure acceptable routes. Such feature would to a large extent eliminate non-acceptable routes.
  - Modifying the sailing path polygons defined as convex hulls to ensure feasible sailing paths around all land areas.
  - Expanding the database of included ports.
- Suggestions for expanding capabilities:
  - Adding capability of generating routes avoiding canals and ECAs based on user preference.
  - Adding the ability to define custom ports on a map or possibly to load port definitions from a file.
  - Expanding the algorithm to include the option of creating routes between several ports.

For the simulation model, the most essential missing work is perhaps an evaluation of the result against system data. Beyond that, there are several means for improving the presented model. The method for speed loss estimation seems to be inaccurate for significant wave heights above four metres, and an alternative method for calculating the attainable speed would therefore be of interest. The following is suggested to improve the functionality of the simulation model:

- Calculating the attainable speed based on the resistance and available power to overcome the resistance.
- Importing wave data from swell and wind waves separately to allow for resistance calculations with the ShipX module made for the original model.
- Finding a method for importing less met-ocean data to the Simulink model to reduce the computational time demand. One possibility is to load data for several smaller areas covering different legs of the route.
- Adding conditions for voluntary speed loss to make the model more realistic.

Bibliography

Arribas, F. P., 2007. Some methods to obtain the added resistance of a ship advancing in waves. Ocean Engineering 34 (7), 946–955.

Bakke, M. Ø., Tenfjord, P. S., 7 2017. Simulation-based analysis of vessel performance during sailing. Master's thesis, NTNU, https://brage.bibsys.no/xmlui/handle/11250/223328, not yet published as of September 2017.

Beeker, E., 2004. An algorithm for finding the shortest sailing distance from any maritime navigable point to a designated port. Mathematical and computer modelling 39 (6-8), 641–647.

Bellman, R., 1958. On a routing problem. Quarterly of applied mathematics 16 (1), 87–90.

Black, J. D., Henson, W. L. W., Jan 2014. Hierarchical load hindcasting using reanalysis weather. IEEE Transactions on Smart Grid 5 (1), 447–455.

Bovet, J., 1986. Une amelioration de la methode de dijkstra pour la recherche d'un plus court chemin dans un reseau. Discrete applied mathematics 13 (1), 93–96.

Chakrabarti, S. K., 2005. Chapter 3 - ocean environment. In: CHAKRABARTI, S. K. (Ed.), Handbook of Offshore Engineering. Elsevier, London, pp. 79 – 131.
URL http://www.sciencedirect.com/science/article/pii/B9780080443812500060

Chang, K. Y., Jan, G., Parberry, I., 09 2003. A method for searching optimal routes with collision avoidance on raster charts. The Journal of Navigation 56, 371 – 384.

Chen, S., Frouws, K., Van De Voorde, E., 2011. Simulation-based optimization of ship design for dry bulk vessels. Maritime Economics & Logistics 13 (2), 190–212.

Chew, P., 1986. There is a planar graph almost as good as the complete graph. In: Proceedings of the second annual symposium on Computational geometry. ACM, pp. 169–177.

Dijkstra, E. W., Dec 1959. A note on two problems in connexion with graphs. Numerische Mathematik 1 (1), 269–271.
URL https://doi.org/10.1007/BF01386390

ECMWF, 2019. Encoding details.
URL https://confluence.ecmwf.int/display/WLW/Encoding+details

Erikstad, S., Grimstad, A., Johnsen, T., Borgen, H., 01 2015. Vista (virtual sea trial by simulating complex marine operations): Assessing vessel operability at the design stage. pp. 107–123.

Fagerholt, K., Heimdal, S. I., Loktu, A., May 2000. Shortest path in the presence of obstacles: An application to ocean shipping. Journal of the Operational Research Society 51 (6).

Fathi, D. E., Grimstad, A., Johnsen, T. A., Nowak, M. P., Stàlhane, M., 2013. Integrated decision support approach for ship design. In: OCEANS-Bergen, 2013 MTS/IEEE. IEEE, pp. 1–8.

Fishman, G. S., 2001. Modeling Concepts. Springer New York, New York, NY, pp. 36–69.
URL https://doi.org/10.1007/978-1-4757-3552-9_2

Garrison, T., 2005. Oceanography : an invitation to marine science, 5th Edition. Brooks/-Cole, Belmont, Calif, Ch. 10, p. 254.

Gosavi, A., 2015. Simulation-based optimization parametric optimization techniques and reinforcement learning.

Grieg Star, 2013. L-class vessels.
URL https://www.griegstar.com/wp-content/uploads/2018/07/Pocket-Planner-L-Class-griegstar_com.pdf

Hart, P. E., Nilsson, N. J., Raphael, B., 1968. A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics 4 (2), 100–107.

Johnson, D. B., Jan. 1977. Efficient algorithms for shortest paths in sparse networks. J. ACM 24 (1), 113.
URL https://doi.org/10.1145/321992.321993

Kauczynski, W. E., 2012. Study of the reliability of the ship transportation by applying the historical hindcast weather data. In: ASME 2012 31st International Conference on Ocean, Offshore and Arctic Engineering. American Society of Mechanical Engineers, pp. 195–201.

Kwon, Y., 2008. Speed loss due to added resistance in wind and waves. Naval Architect, 14–16.

Law, A. M., 2007. Simulation modeling and analysis, 4th Edition. McGraw-Hill series in industrial engineering and management science. McGraw-Hill, Boston.

Li, F., Klette, R., 2011. Convex Hulls in the Plane. Springer London, London, Ch. 4, pp. 93–125.
URL https://doi.org/10.1007/978-1-4471-2256-2_4

Lu, R., Turan, O., Boulougouris, E., Banks, C., Incecik, A., 2015. A semi-empirical ship operational performance prediction model for voyage optimization towards energy efficient shipping. Ocean Engineering 110, 18 – 28, energy Efficient Ship Design and Operations.
URL http://www.sciencedirect.com/science/article/pii/S0029801815003558

MathWorks, 2019a. Mapping toolbox users guide.
URL https://se.mathworks.com/help/pdf_doc/map/map_ug.pdf

MathWorks, 2019b. Matlab primer.
URL https://se.mathworks.com/help/pdf_doc/matlab/getstart.pdf

MathWorks, 2019c. Read global self-consistent hierarchical high-resolution geography.
URL https://www.mathworks.com/help/map/ref/gshhs.html

MathWorks, 2019d. Simevents getting started guide.
URL https://se.mathworks.com/help/pdf_doc/simevents/simevents_gs.pdf

MathWorks, 2019e. Simulink getting started guide.
URL https://se.mathworks.com/help/pdf_doc/simulink/sl_gs.pdf

metoffice.gov.uk, 2020. Beaufort wind force scale.
URL https://www.metoffice.gov.uk/weather/guides/coast-and-sea/beaufort-scale

Möhring, R., Schilling, H., Schütz, B., Wagner, D., Willhalm, T., 02 2006. Partitioning graphs to speedup dijkstra's algorithm. ACM Journal of Experimental Algorithmics 11, 1–29.

Rohnert, H., 1986. Shortest paths in the plane with convex polygonal obstacles. Information Processing Letters 23 (2), 71 – 76.
URL http://www.sciencedirect.com/science/article/pii/0020019086900451

Sargent, R. G., 2005. Verification and validation of simulation models. In: Proceedings of the 37th conference on Winter simulation. winter simulation conference, pp. 130–143.

Searoutes.com, 2020. Bridge the gap with our apis.
URL https://discover.searoutes.com

Steen, S., Minsaas, K., 2014. Ship resistance : Tmr4220 naval hydrodynamics : lecture notes.

Tanaka, M., Kobayashi, K., May 2017. A route generation algorithm for an optimal fuel routing problem between two single ports. International Transactions in Operational Research.

Wessel, P., Smith, W. H. F., 2019. Gshhg, a global self-consistent, hierarchical, high-resolution geography database.
URL https://www.soest.hawaii.edu/pwessel/gshhg

Xu, J. B., Wang, Z. K., Song, L. S., 2012. Study on optimal algorithm for shipping route automatic-generation based on electronic chart. In: Applied Mechanics and Materials. Vol. 105. Trans Tech Publ, pp. 2133–2139.

Appendix A

Electronic appendix

This thesis is submitted with an electronic appendix containing of the MATLAB files for the route-finding algorithm, and the MATLAB and Simulink files for the simulation model.

Appendix B

GSHHG data

Table B.1: Fields describing each element of the structure array containing the GSHHG data, (MathWorks, 2019c; Wessel and Smith, 2019)

Field name	Description
Geometry	Either ”polygon” or ”line”
BoundingBox	[minimum longitude, minimum latitude; maximum longitude, maximum latitude]
Lat	Vector with latitude coordinates for the polygon nodes
Lon	Vector with longitude coordinates for the polygon nodes
South	Southern extreme latitude
North	Northern extreme latitude
West	Western extreme longitude
East	Eastern extreme latitude
Area	Area of polygon in square kilometers
Level	Level 1-6 describes different geographical entities
LevelString	Description of geography describes by polygon. Empty for levels five and six
NumPoints	Number of nodes in polygon
FormatVersion	Version of file format
Source	Source of data. Either ”WVS” (World Vector Shorelines) or ”WDBII” (CIA World DataBank II)
CrossesGreenwich	Logical variable, 0 or 1 if false or true.
CrossesDateline	Logical variable, 0 or 1 if false or true.
GSHHS_ID	Unique polygon id number
RiverLake	Logical indicator, true only if level 2 polygon is the fat part of a major river.
AreaFull	Area of full-resolution polygon []
Container	ID of polygon enclosing the current polygon. If not relevant the value is -1
Ancestor	ID of polygon in the full resolution data that was the source of the current polygon. Equals -1 if none.

Appendix C

Modified shoreline sailing path polygons

Table C.1 lists the shoreline polygons with ID 0-30, and the third column indicates if the corresponding shoreline sailing path polygon has been modified. The additional shoreline polygons that have modified sailing path polygons are also included. When the shoreline sailing path polygons have not been modified the shoreline sailing paths are defined as convex hulls around the shoreline polygon.

Table C.1: List showing the shoreline sailing path polygons that have been modified.

ID	Index	Modified	Land area name
0	1	Yes	Eurasia
1	2	Yes	Africa
2	3	Yes	North America
3	4	Yes	South America
4	5702	Yes	Antarctic ice sheet
5	[]	Not included	Antarctic land mass
6	5	Yes	Australia
7	6	Yes	Greenland
8	7	Yes	New Guinea
9	8	Yes	Borneo
10	9	No	Madagascar
11	10	No	Baffin Island (Arctic Archipelago)
12	11	Yes	Sumatra
13	12	Yes	Honshu (Yespan)
14	13	No	Victoria Island (Arctic Archipelago)
15	14	Yes	Great Britain
16	15	No	Ellesmere Island (Arctic Archipelago)
17	16	No	Sulawesi (Indonesia)
18	17	Yes	South Island (New Zealand)
19	18	No	Yesva
20	19	No	North Island (New Zealand)
21	20	No	Newfoundland
22	21	No	Cuba
23	22	Yes	Luzon (Philippines)
24	23	No	Iceland
25	24	No	Mindanao (Philippines)
26	25	No	Ireland
27	26	Yes	Hokkaido (Yespan)
28	27	No	Sakhalin (Russia)
29	28	Yes	Hispaniola
30	29	No	Banks Island (Arctic Archipelago)
35	33	Yes	Isla Grande de Tierra del Fuego
43	40	Yes	Kuamoto (Yespan)
70	67	Yes	Bangka (Indonesia)
71	68	Yes	Palawan (Philippines)
148	138	Yes	Santa Ins Island (Chile)
274	254	Yes	Aracena Island (Chile)
280	259	Yes	Clarence Island (Chile)
417	382	Yes	Londonderry Island (Chile)
618	566	Yes	Geojedo (South Korea)
736	674	Yes	Middle Caicos (Turks and Caicos Islands)
927	850	Yes	East Caicos (Turks and Caicos Islands)
4419	3938	Yes	Baisha Main Island (Taiwan)

Appendix D

MATLAB code for the route generating algorithm

D.1 Main script and main functions

D.1.1 RouteGenerator.m

```
%% RouteGenerator
% Script for automatic route generation between two ports

% Author:   Ole Brynjar Helland Paulsen
% Date:     08.02.2020

clear variables
close all
addpath(genpath('Functions'))    % Path for functions used in this ...
    script
addpath('Input')                 % Path for shoreline data

% Coordinates of the origin and destination ports
[ports, port_path, origin, destination] = select_ports();

% Get max length of each leg (nm), and option for sailing in polar ...
    waters.
% Polar = 1, when polar sailing is allowed
[max_length, polar] = user_input();

% Generate route
path = GetRoute(ports, port_path, origin, destination, max_length, ...
    polar);

% Create maps
```

```
disp('Generating maps..')
[f1, f2, H1, H2] = draw_maps(path);

% User verification. Gives the option to alter the route by adding a
% waypoint, or by changing the maximum length of the sailing legs. ...
    In any
% case a option to save the route is given.
f = msgbox({'Please inspeckt the route, then press OK to proceed', ...
    'Verify route'});
waitfor(f)
answer = questdlg('If the route is not satifactory, try adding a ...
    waypoint or alter the length of the sailing legs',...
    'Keep route?', 'Save current route','Alter leg length', ...
    'Add waypoints', 'Save current route');
switch answer
    case 'Save current route'
        % Save route
        uisave('path','route');
    case 'Alter leg length'
        % Gather new max leg length
        prompt = {'Enter desired maximum lenght of sailing legs [nm]'};
        title = 'Length sailing legs';
        in = inputdlg(prompt,title,[1 40]);
        max_length = str2double(in{1});

        % Generate new route
        path = GetRoute(ports, port_path, origin, destination, ...
            max_length, polar);

        % Remove old route from maps and display the new
        figure(f1)
        delete(H1)
        geoshow([path.Lat],[path.Lon],'color','red')

        figure(f2)
        delete(H2)
        geoshow([path.Lat],[path.Lon],'color','red')

        % Save route
        answer2 = questdlg('Save route?',...
            'Save', ...
            'Save','Cancel', 'Save');
        if strcmp(answer2,'Save')
            uisave('path','route')
        end
    case 'Add waypoints'
        % Add waypoint on map
        an = questdlg('Please click on one or several suitable ...
            wayopints, and then press return.', ...
            'Add waypoints','Ok','Ok');
        % Input
        [lat, lon] = inputm();

        % Define new set of ports with waypoint coordinates
        ports1 = [ports(1,:); [lat(1), lon(1)]];
        dest1 = 'the first waypoint';
        ports2 = [[lat(end), lon(end)]; ports(2,:)];
```

```
76          orig2 = 'the last waypoint';
77
78          % Port paths
79          port_path1(1) = port_path(1);
80          port_path1(2).Lat = [];
81          port_path1(2).Lon = [];
82          port_path2(1).Lat = [];
83          port_path2(1).Lon = [];
84          port_path2(2) = port_path(2);
85
86          % Generate two seperate routes
87          path1 = GetRoute(ports1, port_path1, origin, dest1, ...
                max_length, polar);
88          path2 = GetRoute(ports2, port_path2, orig2, destination, ...
                max_length, polar);
89
90          % Merge paths
91          Latpts = [path1.Lat(1:end-1), transpose(lat), ...
                path2.Lat(2:end)];
92          Lonpts = [path1.Lon(1:end-1), transpose(lon), ...
                path2.Lon(2:end)];
93
94          % Update route data
95          [course, distnm] = legs(Latpts,Lonpts,'rh');
96
97          path.Lat = Latpts;
98          path.Lon = Lonpts;
99          path.Course = course;
100         path.Distnm = distnm;
101         path.Tot_dist = sum(path.Distnm);
102
103         % Longitude coordinates
104         lons = path.Lon;
105
106         % Find if route has positive and negative longitude coordinates
107         a = find(diff(lons>0)≠0);
108         % Sum of absoulute value of coordinates before and after ...
                sign change.
109         b = abs(lons(a)) + abs(lons(a+1));
110         % If the sum two coordinates are above 180 dregress, the ...
                route goes
111         % across the international date line.
112         if max(b) > 180
113             path.Cross_date = 1;
114             % Indecies of cross date coordinates in sign change ...
                    index vector
115             c = find(b>180);
116             % Index of node prior to fist crossing of date line
117             d = a(c);
118             path.Cross_ind = d;
119         else
120             path.Cross_date = 0;
121             path.Cross_ind = [];
122         end
123
124         % Remove old route from maps and display the new
125         figure(f1)
```

```
126         delete(H1)
127         geoshow([path.Lat],[path.Lon],'color','red')
128 
129         figure(f2)
130         delete(H2)
131         geoshow([path.Lat],[path.Lon],'color','red')
132 
133         % Save route
134         answer3 = questdlg('Save route?',...
135             'Save', ...
136             'Save','Cancel', 'Save');
137         if strcmp(answer3,'Save')
138             uisave('path','route')
139         end
140 end
```

D.1.2 select_ports.m

```
1  function [ports, port_path, origin, destination] = select_ports()
2  %% Function for defining origin and destination port. Ports are either
3  % selected from a list of included ports, or defined manually by
4  % coordinates. The included ports can have a "port path" from the ...
       port to
5  % open waters. This is not an option when manually defining ports.
6  
7  % Author:   Ole Brynjar Helland Paulsen
8  % Date:     01.03.2020
9  load ports_db.mat  ports_db
10 
11 % Preallocate variables
12 ports= zeros(2,2);
13 port_path(2).Lon = 9;
14 %% Define origin port
15 
16 % List of port names
17 list = [ports_db.Name];
18 % Index list
19 list_ind = 1:length(ports_db);
20 
21 % Prompt user input
22 [indx1,tf1] = listdlg('PromptString','Select origin port:',...
23     'SelectionMode','single','ListSize',[150,170],'ListString',list,...
24     'CancelString','Manual input');
25 
26 if tf1 == 1
27     % Port choosen from list
28     % Port coordinates for the route-finding algorithm
29     ports(1,:) = [ports_db(indx1).Lat(end), ports_db(indx1).Lon(end)];
30     % Paths from port to open seas
31     port_path(1).Lat = ports_db(indx1).Lat;
32     port_path(1).Lon = ports_db(indx1).Lon;
33     % Port name
34     name  = ports_db(indx1).Name;
35     origin = name{1};
```

```
36      % Exclude origin port from list
37      list(indx1) = [];
38      list_ind(indx1) = [];
39  else
40      % Custom port. Ports must have open acess to the seas
41      % Read input
42      prompt = {'Latitude:','Longitude:', 'Name:'};
43      title = 'Origin port';
44      dims = [1 12; 1 12; 1 45];
45      x = inputdlg(prompt,title,dims);
46      lat = x(1);
47      Lat = str2double(lat{1});
48      lon = x(2);
49      Lon = str2double(lon{1});
50      % Port coordinates
51      ports(1,:) =[Lat, Lon];
52      % Path from port to open seas not supported, set to void
53      port_path(1).Lat = [];
54      port_path(1).Lon = [];
55      % Port name
56      name  = x(3);
57      origin = name{1};
58  end
59
60  %% Define destination port
61
62  % Prompt user input
63  [indx,tf2] = listdlg('PromptString','Select destination port:',...
64      'SelectionMode','single','ListSize',[150,170],'ListString',list,...
65      'CancelString','Manual input');
66
67  if tf2 == 1
68      % Port from list
69      indx2 = list_ind(indx);
70      ports(2,:) = [ports_db(indx2).Lat(end), ports_db(indx2).Lon(end)];
71      port_path(2).Lat = fliplr(ports_db(indx2).Lat);
72      port_path(2).Lon = fliplr(ports_db(indx2).Lon);
73      name  = ports_db(indx2).Name;
74      destination = name{1};
75  else
76      % Custom port
77      prompt = {'Latitude:','Longitude:', 'Name:'};
78      title = 'Destination port';
79      dims = [1 12; 1 12; 1 45];
80      x = inputdlg(prompt,title,dims);
81      lat = x(1);
82      Lat = str2double(lat{1});
83      lon = x(2);
84      Lon = str2double(lon{1});
85      ports(2,:) =[Lat, Lon];
86      port_path(2).Lat = [];
87      port_path(2).Lon = [];
88      name  = x(3);
89      destination = name{1};
90  end
91  end
```

D.1.3 user_input.m

```
function [max_length, polar] = user_input()
%% Function for setting user prefeances.

% Author:   Ole Brynjar Helland Paulsen
% Date:     17.06.2018

prompt = {'Enter desired maximum lenght of sailing legs [nm]'};
title = 'Length sailing legs';
in = inputdlg(prompt,title,[1 40]);
max_length = str2double(in{1});

answer = questdlg('Include routes in polar waters?', ...
    'Polar sailing', ...
    'Yes','No', 'Not relevant','No');

% Handle response
switch answer
    case 'Yes'
        polar = 1;
    case 'No'
        polar = 0;
    case 'Not relevant'
        polar = 0;
end

end
```

D.1.4 GetRoute.m

```
function final_path = GetRoute(ports, port_path, startport, ...
    nextport, max_length_legs, polar)
%% Function for generating routes between two ports
% Author:   Ole Brynjar Helland Paulsen
% Date:     01.05.2020

% Max lenght of sailing legs
global max_length;
max_length = max_length_legs;

% Store origin and destination port as part of result
final_path.Origin = startport;
final_path.Destination = nextport;

% Variable indicating if route circumvent Antarctica
antarctic_path = 0;

%% Inital route

% Compute waypoints for route divided in n legs of the given ...
    maximum length
```

```
20 [lat_org, lon_org] = ...
       waypoints(ports(1,1),ports(1,2),ports(2,1),ports(2,2));
21
22 %% Limits for maps and shoreline polygons
23
24 % Find min and max longitude coordinates
25 min_lon = min(ports(2,2),ports(1,2));
26 max_lon = max(ports(2,2),ports(1,2));
27
28 % Min and max latitude of initial route
29 min_lat = min(lat_org);
30 max_lat = max(lat_org);
31
32 if abs(max_lon - min_lon) > 180
33     %Ports lie on opposite side of the date line
34     cross_date = 1;
35     % The shapefile boundingbox demands that min < max. Must therefore
36     % load the complete longitudinal span
37     blon_min = -180;
38     blon_max = 180;
39 else
40     % Ports are located on the same side of the date line
41     cross_date = 0;
42     %Longitude limits for shoreline polygons to be loaded
43     blon_min = (min_lon-10);
44     blon_max = (max_lon+10);
45 end
46
47 %% Shoreline data
48 % Loading shorelines within the area set by the bounding box
49 % Path for input data must have been added to the directory of Matlab
50
51 %Bounding box
52 bbox =[blon_min, (min_lat-10);
53        blon_max, (max_lat+10)];
54
55 % Low resoulution shorelines
56 shorelines = shaperead('shorelines.shp','BoundingBox',bbox);
57
58 % Modified hulls as shoreline sailing path polygons
59 hull = shaperead('sailing_paths.shp','BoundingBox',bbox);
60
61 %% Intersection test
62 % Check if the initial route intersects any land area polygons, and ...
       if so
63 % return the correct ID.
64 [ID] = polycheck(shorelines, lat_org, lon_org, startport, nextport,...
65     hull, ports, cross_date);
66
67 %% Generate route
68 if isempty(ID) == 1
69     % If route does not intersect land areas, the great cirlce ...
           route is the
70     % final path.
71     final_path.Lat = transpose(lat_org);
72     final_path.Lon = transpose(lon_org);
73 else
```

```
74      %% Fetch shorline polygon and sailing path polygon for the ...
            first land
75      % area to circumvent
76
77      % Find index of hull struct that correspond to the correct ID
78      index_h = find([hull.ID] == ID);
79
80      %Latitude and longitude coordinates of the given convex hull
81      hull_lat = hull(index_h).Y;
82      hull_lon = hull(index_h).X;
83
84      %Remove NaN entries
85      hull_lat(isnan(hull_lat)) = [];
86      hull_lon(isnan(hull_lon)) = [];
87
88      %Number of different nodes in the hull polygon. First node is ...
            repeated
89      %in the end
90      num_nodes = length(hull_lat)-1;
91
92      %Index of shoreline polygon with the desired ID. In some cases
93      %not equal to the index of the same hull polygon
94      index = find([shorelines.GSHHS_ID] == ID);
95
96      %Find latitude and longitude of the shoreline polygons
97      shore_lat = shorelines(index).Y;
98      shore_lon = shorelines(index).X;
99
100     % Latitudes and longitudes of ports
101     port_lats = [ports(1,1), ports(2,1)];
102     port_lons = [ports(1,2), ports(2,2)];
103
104     %% Find suitable connection nodes for the predefined sailing path
105     % polygon
106
107     if ID == 4 % Connection nodes for Antartic continent
108
109         if polar == 0
110             % Alter coordinates of antatica hull(ID = 4, index = ...
                    5584) in
111             % order to restrict acess.
112             hull_lon = [-179.999, -165:15:165, 179.999];
113             num_nodes = length(hull_lon)-1;
114             hull_lat =1:(num_nodes+1);
115             hull_lat(:) = -60;
116             antarctic_path = 1;
117         end
118         [connect_nodes] = antarctic(hull_lat, hull_lon, lat_org, ...
                lon_org, polar, cross_date);
119
120         % Check if ports lie on border of modified hull
121         [¬, on] = inpolygon(port_lats, port_lons, hull_lat, hull_lon);
122     else
123         if cross_date == 1
124             %When the route is crossing the dateline,the port on the
125             %opposite side from the obstacle polygon must be ...
                    modified to
```

```
126             %have equal sign as the longitude coordinates of the ...
                    obstacle
127             %polygon. This is in order to create the correct convex ...
                    hull
128             %and thus get the correct connection nodes.
129             ports_adj = adjust_ports(ports, hull_lon);
130
131             disp('The port coordinates are adjusted to:')
132             disp(ports_adj)
133
134             % Latitudes and longitudes of ports
135             port_lats = [ports_adj(1,1), ports_adj(2,1)];
136             port_lons = [ports_adj(1,2), ports_adj(2,2)];
137         else
138             ports_adj = ports;
139         end
140
141         %Convex hull around current land polygon
142         conv = convhull(hull_lat, hull_lon);
143         conv_lat = hull_lat(conv);
144         conv_lon = hull_lon(conv);
145
146         % Check if any of the ports are located inside a convex hull
147         % around the land polygon
148         in_conv = inpolygon(port_lats, port_lons, conv_lat, conv_lon);
149
150         if sum(in_conv) == 0
151             % None of the ports are nodes in the current sailing path
152             % polygon
153             on = [0,0];
154             % Check if ports are visible to each other with respect ...
                    to the
155             % sailing path polygon
156             [¬, ¬, inter_points] = inter_coords(port_lats, ...
                    port_lons, hull_lat,...
157             hull_lon, cross_date);
158
159             if inter_points > 0
160                 % Both port are outside the convex hull and not ...
                        visible to each
161                 % other. 4 connection nodes returned
162                 [connect_nodes] = ...
                        connecting_outside_non_visi(hull_lat, ...
                        hull_lon, ports_adj);
163             else
164                 % Both ports lie outside the convex hull, but are ...
                        visible to
165                 % each other. Either 2 or 4 connection nodes returned.
166                 [connect_nodes, dir, inport] = ...
                        connecting_outside_visi(hull_lat, hull_lon, ...
                        ports_adj, lat_org, lon_org, cross_date);
167             end
168         else
169             % Check if ports are visible from each other, by a ...
                    straight line, with
170             % regards to the given shoreline polygon
```

```
171             [¬, ¬, inter_points] = inter_coords(port_lats, ...
                    port_lons, shore_lat,...
172                 shore_lon, cross_date);
173
174             % Check if ports lie inside modified hull
175             [in, on] = inpolygon(port_lats, port_lons, hull_lat, ...
                    hull_lon);
176
177             if inter_points > 0 && sum(in_conv) == 2
178                 % Both ports are inside the convex hull and not ...
                        visible to each
179                 % other. Either 2 or 4 connection nodes returned.
180                 [connect_nodes, dir, inport] = ...
                        connecting_non_visi_inside(hull_lat, hull_lon, ...
                        port_lats, port_lons, num_nodes);
181             elseif inter_points > 0
182                 % One port is inside the convex hull and the ports ...
                        are not
183                 % visible to each other. Either 2 or 4 connection ...
                        nodes returned.
184                 [connect_nodes, dir, inport] = ...
                        connecting_non_visi_in_out(hull_lat, hull_lon, ...
                        port_lats, port_lons, num_nodes, in_conv);
185             else
186                 %Ports are visible to each other by a straight line.
187                 [connect_nodes, dir, inport] = ...
                        connecting_visi(hull_lat, hull_lon, port_lats, ...
                        port_lons, num_nodes, lat_org, lon_org, ...
                        cross_date, in_conv, in, on);
188             end
189         end
190     end
191
192     %% Check if port is node in the sailing path polygon
193
194     % Variable indicating if port is a hull node
195     port_node = [0, 0];
196
197     if sum(on) ≥ 1
198         % At least on port lie on border of modified hull
199         for port = 1:2
200             % Check if port is a node in the current modified hull
201             lat_diff = abs(hull_lat - port_lats(port));
202             [min_lat, i_lat] = min(lat_diff);
203             lon_diff = abs(hull_lon - port_lons(port));
204             [min_lon, i_lon] = min(lon_diff);
205
206             if min_lat == 0 && min_lon == 0 && i_lat == i_lon
207                 %Port is a node of the modified hull
208                 port_node(port) = 1;
209             end
210         end
211     end
212
213     %% Generate route route circumventing the intersected land area
214
215     % Path around initial intersecting land area
```

```
216     if size(connect_nodes,1) == 1
217         fprintf('One path omitting land polygon ID %d is ...
                generated.\n\n',ID);
218         [path, min_path, ¬, connect_nodes] = one_path(ports, ...
                hull_lat, hull_lon, shore_lat, shore_lon, ...
                connect_nodes, num_nodes, inport, port_node, dir);
219     else
220         fprintf('Two paths omitting land polygon ID %d is ...
                generated.\n\n',ID);
221         [path, min_path, ¬, connect_nodes] = two_paths(ports, ...
                hull_lat, hull_lon, shore_lat, shore_lon, ...
                connect_nodes, num_nodes, polar, port_node);
222     end
223
224     % Struct for storing the coordinates of the final path
225     final_path.Lat = path(min_path).Lat;
226     final_path.Lon = path(min_path).Lon;
227
228     %% Check if it is necessary to load shoreline data anew, to ...
            cover a
229     % larger area
230
231     % New bounding limits
232     lons = final_path.Lon;
233     [cross_date, ¬] = cross_date_line(lons);
234
235     if cross_date == 1
236         blon_min = -180;
237         blon_max = 180;
238     else
239         blon_min = min(lons) - 10;
240         blon_max = max(lons) + 10;
241     end
242
243     blat_min = min([bbox(1,2), min(final_path.Lat)]);
244     blat_max = max([bbox(2,2), max(final_path.Lat)]);
245
246     %New bounding box
247     bbox_new = [blon_min, blat_min;
248                 blon_max, blat_max];
249
250     % Check if equal
251     if isequal(bbox_new, bbox) == 0
252         clear shorelines hull
253         % Low resoulution shorelines
254         shorelines = ...
                shaperead('shorelines.shp','BoundingBox',bbox_new);
255
256         % Modified hulls
257         hull = shaperead('sailing_paths.shp','BoundingBox',bbox_new);
258     end
259
260
261     %% Initialise list of dummy ports
262     % The dummy ports corresponds to the connection nodes of the
263     % the previous path
264
```

```
265     dummy_port_list = zeros(50,4);
266
267     % The inital entries of the list of dummy ports, are the ports ...
            and the
268     % connect nodes of the inital obstacle polygon.
269     dummy_port_list(1,:) = ...
270         [ports(1,1), ports(1,2), ...
                hull_lat(connect_nodes(min_path,1)), ...
271         hull_lon(connect_nodes(min_path,1))];
272
273     dummy_port_list(2,:) = [hull_lat(connect_nodes(min_path,2)),...
274         hull_lon(connect_nodes(min_path,2)), ports(2,1), ports(2,2)];
275
276     %% Initialize iterations
277     num_rows = 2;   % Number of rows of dummy ports
278     startport = 0;
279     nextport = 1;
280     dummy_orig = 1:2:100;
281     dummy_dest = 2:2:100;
282     i = 1;
283
284     % The following routine runs through the list of dummy ports to ...
            check
285     % each distance for new intersections:
286
287     % Break loop if it uses more than 120 seconds
288     time0 = tic;
289     timeLimit = 120;
290     while i ≤ num_rows && toc(time0)<timeLimit
291         startport = startport + 1;
292         nextport = nextport + 1;
293
294         % Dummy ports of the current distance
295         dummy_ports = [dummy_port_list(i,1),dummy_port_list(i,2);
296             dummy_port_list(i,3),dummy_port_list(i,4)];
297
298         dummy_ports(:,2) = wrapTo180(dummy_ports(:,2));
299
300         % Great circle route divided in waypoints
301         [latpts,lonpts] = ...
                waypoints(dummy_ports(1,1),dummy_ports(1,2),...
302             dummy_ports(2,1),dummy_ports(2,2));
303
304         % Adjust longitudes to between [-180, 180]
305         lonpts = wrapTo180(lonpts);
306
307         % Find if route is crossing the date line
308         [cross_date, ¬] = cross_date_line(lonpts);
309
310         % Check if route intersects land areas
311         origin = ['dummy port ', num2str(dummy_orig(i))];
312         destination = ['dummy port ', num2str(dummy_dest(i))];
313
314         [ID] = polycheck(shorelines, latpts, lonpts, origin, ...
                destination, hull, dummy_ports, cross_date);
315
316         if isempty(ID) == 0
```

```
317             %% Fetch the shorline polygon and sailing path polygon ...
                    for the
318             % current land area
319
320             % If path between dummy ports intersect land, find ...
                    shortes path
321             % around land area, and update the dummy port matrix ...
                    with new
322             % coordinates
323
324             % Find index of hull struct that correspond to the ...
                    correct ID
325             index_h = find([hull.ID] == ID);
326
327             %Latitude and longitude coordinates of the given convex ...
                    hull
328             hull_lat = hull(index_h).Y;
329             hull_lon = hull(index_h).X;
330
331             %Remove NaN entries
332             hull_lat(isnan(hull_lat)) = [];
333             hull_lon(isnan(hull_lon)) = [];
334
335             %Number of different nodes in the hull polygon. First ...
                    node is
336             %repeated in the end
337             num_nodes = length(hull_lat)-1;
338
339             %Index of shoreline polygon with the desired ID. In ...
                    some cases
340             %not equal to the index of the same hull polygon
341             index = find([shorelines.GSHHS_ID] == ID);
342
343             %Find latitude and longitude of the shoreline polygons
344             shore_lat = shorelines(index).Y;
345             shore_lon = shorelines(index).X;
346
347             % Latitudes and longitudes of dummy ports
348             port_lats = [dummy_ports(1,1), dummy_ports(2,1)];
349             port_lons = [dummy_ports(1,2), dummy_ports(2,2)];
350
351             %% Establish suitable connection nodes
352             if ID == 4
353                 if polar == 0
354                     % Alter coordinates of antatica hull(ID = 4,
355                     % index = 5702) in order to restrict acess.
356                     hull_lon = [-179.999, -165:15:165, 179.999];
357                     num_nodes = length(hull_lon)-1;
358                     hull_lat =1:(num_nodes + 1);
359                     hull_lat(:) = -60;
360                     antarctic_path = 1;
361                 end
362                 [connect_nodes] = antarctic(hull_lat, hull_lon, ...
                        latpts, lonpts, polar, cross_date);
363
364                 % Check if ports lie on border of modified hull
```

```
365                     [¬, on] = inpolygon(port_lats, port_lons, hull_lat, ...
                            hull_lon);
366
367                 else
368                     if cross_date == 1
369                         %When the route is crossing the dateline,the ...
                                port on
370                         %the opposite side from the obstacle polygon ...
                                must be
371                         %modified to have equal sign as the longitude
372                         %coordinates of the obstacle polygon. This is ...
                                in order
373                         %to create the correct convex hull and thus get the
374                         %correct connection nodes.
375
376                         ports_adj = adjust_ports(dummy_ports, hull_lon);
377
378                         disp('The port coordinates are adjusted to:')
379                         disp(ports_adj)
380
381                         % Latitudes and longitudes of ports
382                         port_lats = [ports_adj(1,1), ports_adj(2,1)];
383                         port_lons = [ports_adj(1,2), ports_adj(2,2)];
384
385                     else
386                         ports_adj = dummy_ports;
387                     end
388
389                     %Convex hull around the current land polygon
390                     conv = convhull(hull_lat, hull_lon);
391                     conv_lat = hull_lat(conv);
392                     conv_lon = hull_lon(conv);
393
394                     % Check if any of the ports are located inside a convex
395                     % hull around the land polygon
396                     in_conv = inpolygon(port_lats, port_lons, conv_lat, ...
                            conv_lon);
397
398                     if sum(in_conv) == 0
399                         % None of the ports are nodes in the current ...
                                sailing path
400                         % polygon
401                         on = [0,0];
402                         % Check if ports are visible to each other with ...
                                respect to the
403                         % sailing path polygon
404                         [¬, ¬, inter_points] = inter_coords(port_lats, ...
                                port_lons, hull_lat,...
405                         hull_lon, cross_date);
406
407                         if inter_points > 0
408                             % Both port are outside the convex hull and ...
                                    not visible to each
409                             % other. 4 connection nodes returned
410                             [connect_nodes] = ...
                                    connecting_outside_non_visi(hull_lat, ...
                                    hull_lon, ports_adj);
```

```
411                         else
412                             % Both ports lie outside the convex hull, ...
                                    but are visible to
413                             % each other. Either 2 or 4 connection ...
                                    nodes returned.
414                             [connect_nodes, dir, inport] = ...
                                    connecting_outside_visi(hull_lat, ...
                                    hull_lon, ports_adj, latpts, lonpts, ...
                                    cross_date);
415                         end
416                     else
417                         %Check if ports are visible from each other, by ...
                                a straight line, with
418                         %regards to the given shoreline polygon
419                         [¬, ¬, inter_points] = inter_coords(port_lats, ...
                                port_lons, shore_lat,...
420                             shore_lon, cross_date);
421
422                         % Check if ports lie inside modified hull
423                         [in, on] = inpolygon(port_lats, port_lons, ...
                                hull_lat, hull_lon);
424
425                         if inter_points > 0 && sum(in_conv) == 2
426                             % Both ports are inside the convex hull and ...
                                    not visible to each
427                             % other. Either 2 or 4 connection nodes ...
                                    returned.
428                             [connect_nodes, dir, inport] = ...
                                    connecting_non_visi_inside(hull_lat, ...
                                    hull_lon, port_lats, port_lons, ...
                                    num_nodes);
429                         elseif inter_points > 0
430                             % One port is inside the convex hull and ...
                                    the ports are not
431                             % visible to each other.
432                             [connect_nodes, dir, inport] = ...
                                    connecting_non_visi_in_out(hull_lat, ...
                                    hull_lon, port_lats, port_lons, ...
                                    num_nodes, in_conv);
433                         else
434                             %Ports are visible to each other by a ...
                                    straight line.
435                             [connect_nodes, dir, inport] = ...
                                    connecting_visi(hull_lat, hull_lon, ...
                                    port_lats, port_lons, num_nodes, ...
                                    latpts, lonpts, cross_date, in_conv, ...
                                    in, on);
436                         end
437                     end
438                 end
439
440                 %% Check if port is node in modified hull
441
442                 % Variable indicating if port is a hull node
443                 port_node = [0, 0];
444
445                 if sum(on) ≥ 1
```

```
446                     % At least on port lie on border of modified hull
447
448                     for port = 1:2
449                         % Check if port is a node in the current ...
                                modified hull
450                         lat_diff = abs(hull_lat - port_lats(port));
451                         [min_lat, i_lat] = min(lat_diff);
452                         lon_diff = abs(hull_lon - port_lons(port));
453                         [min_lon, i_lon] = min(lon_diff);
454
455                         if min_lat == 0 && min_lon == 0 && i_lat == i_lon
456                             %Port is a node of the modified hull
457                             port_node(port) = 1;
458                         end
459                     end
460                 end
461
462                 %% Find path circumventing the land area
463                 if size(connect_nodes,1) == 1
464                     fprintf('One path omitting land polygon ID %d is ...
                            generated.\n\n',ID);
465                     [path, min_path, ¬, connect_nodes] = ...
                            one_path(dummy_ports, hull_lat, hull_lon, ...
                            shore_lat, shore_lon, connect_nodes, ...
                            num_nodes, inport, port_node, dir);
466                 else
467                     fprintf('Two paths omitting land polygon ID %d is ...
                            generated.\n\n',ID);
468                     [path, min_path, ¬, connect_nodes] = ...
                            two_paths(dummy_ports, hull_lat, hull_lon, ...
                            shore_lat, shore_lon, connect_nodes, ...
                            num_nodes, polar, port_node);
469                 end
470
471                 %% Update final path and dummy port list
472
473                 ind_start = find(final_path.Lon == dummy_ports(1,2));
474                 ind_end = find(final_path.Lon == dummy_ports(2,2));
475                 if isempty(ind_start) == 1 || isempty(ind_end) == 1
476                     disp(i)
477                 end
478
479                 path_end = size(path(min_path).Lat,2);
480                 final_path_end = size(final_path.Lat,2);
481
482                 % Update final path
483                 final_path.Lat = [final_path.Lat(1:(ind_start)),...
484                     path(min_path).Lat(2:(path_end-1)),...
485                     final_path.Lat(ind_end:final_path_end)];
486
487                 final_path.Lon = [final_path.Lon(1:(ind_start)),...
488                     path(min_path).Lon(2:(path_end-1)),...
489                     final_path.Lon(ind_end:final_path_end)];
490
491                 % Update dummy port list
492                 rows = num_rows-i;
493
```

```
494             if num_rows ≠ i
495             dummy_port_list(i+3:i+rows+2,:)...
496                 = dummy_port_list(i+1:num_rows,:);
497             end
498             dummy_port_list(i+1,:) = [dummy_ports(1,1), ...
                    dummy_ports(1,2)...
499                 , hull_lat(connect_nodes(min_path,1)),...
500                 hull_lon(connect_nodes(min_path,1))];
501
502             dummy_port_list(i+2,:) = ...
503                 [hull_lat(connect_nodes(min_path,2))...
504                 hull_lon(connect_nodes(min_path,2)), ...
505                 dummy_ports(2,1), dummy_ports(2,2)];
506
507             num_rows = num_rows + 2;
508         end
509         % Update counter
510         i=i+1;
511     end
512 end
513
514 %% Assemble complete route
515
516 % Include predefined paths to anf from ports if applicable
517 if length(port_path(1).Lat) > 1
518     final_path.Lat  = [port_path(1).Lat(1:end-1), final_path.Lat];
519     final_path.Lon  = [port_path(1).Lon(1:end-1), final_path.Lon];
520 end
521 if length(port_path(2).Lat) > 1
522     final_path.Lat  = [final_path.Lat, port_path(2).Lat(2:end)];
523     final_path.Lon  = [final_path.Lon, port_path(2).Lon(2:end)];
524 end
525
526 % Check the final route goes across the international date line.
527 [cross_date, ind] = cross_date_line(final_path.Lon);
528
529 % Compute headings and distances for the waypoint legs
530 [course, distnm] = legs(final_path.Lat,final_path.Lon,'rh');
531
532 if polar == 0
533     % Polar sailing is not allowed, check if route violate restriction
534     artic = find(final_path.Lat > 70, 1);
535     antarc_ind = find(final_path.Lat < -60);
536     if isempty(artic) == 0
537         % Final route includes artic latitudes, display warning
538         disp('Warning: Route includes latitudes above 70 degrees ...
                North');
539     elseif isempty(antarc_ind) == 0 && antarctic_path == 0
540         % Route includes antarctic latitudes. Adjust path to keep ...
                above -60
541         % degrees south.
542         % Path coordinates
543         lats = transpose(final_path.Lat);
544         lons = transpose(final_path.Lon);
545
546         % Rhumb line at -60 latitude
547         line_lat = [-60, -60];
```

```
548             line_lon = [-180, 180];
549
550             % Points of intersection
551             [lat, lon, ¬] = inter_coords(lats, lons, line_lat, ...
                    line_lon, cross_date);
552
553             % Limit latitudes to -60 degrees
554             final_path.Lat(antarc_ind) = -60;
555
556             % New path includes points of intersection
557             ind_start = antarc_ind(1);
558             ind_end = antarc_ind(end);
559             final_path.Lat = [final_path.Lat(1:ind_start-1),lat(1), ...
                    final_path.Lat(ind_start:ind_end), lat(2), ...
                    final_path.Lat(ind_end+1:end)];
560             final_path.Lon = [final_path.Lon(1:ind_start-1),lon(1), ...
                    final_path.Lon(ind_start:ind_end), lon(2), ...
                    final_path.Lon(ind_end+1:end)];
561
562             % Update date line info
563             [cross_date, ind] = cross_date_line(final_path.Lon);
564         end
565     end
566
567     % Store route information in path struct:
568     % Course and distance
569     final_path.Course = course;
570     final_path.Distnm = distnm;
571
572     % Total distance
573     final_path.Tot_dist = sum(distnm);
574
575     % Crossing date line info
576     final_path.Cross_date = cross_date;
577     final_path.Cross_ind = ind;
578     end
```

D.1.5 draw_maps.m

```
1   function [f1, f2, H1, H2] = draw_maps(path)
2   %% This function draws two maps, of mercantor projection and one ...
        orthogonal
3   % projection, of the route desribed by the "path" variable given as ...
        input.
4   % Author:   Ole Brynjar Helland Paulsen
5   % Date:     17.02.2020
6
7   % Load shoreline data
8   load map_data.mat map_data
9
10  % Define leves
11  levels = [map_data.Level];
12
13  L1 = (levels == 1);
```

```
land = map_data(L1);
L2 = (levels == 2);
lake=map_data(L2);
L3 = (levels == 3);
island = map_data(L3);
L4 = (levels == 4);
pond=map_data(L4);
L5 = (levels == 5);
ice=map_data(L5);
L6 = (levels == 6);
antartic=map_data(L6);

% Latitude coordinates
lats = path.Lat;
% Longitude coordinates
lons = path.Lon;

% Port coordinates
ports = [lats(1), lons(1);
    lats(end), lons(end)];

%% Setting latitude and longitude limits for the maps
%Minimum and maximum latitude of paths
min_lat = min(lats);
max_lat = max(lats);

%Calculate latitude midpoint
lat_origo = min_lat + (max_lat - min_lat)/2;

if path.Cross_date == 0
    % The route is not crossing the date line
    min_lon = min(lons);
    max_lon = max(lons);

    % Longitudinal span
    lon_span = max_lon - min_lon;

    % Origo of orthogonal map
    lon_origo = min_lon + lon_span/2;

else
    % The route goes across the international date line
    % Index of node(s) prior to crossing
    ind = path.Cross_ind;

    if length(ind) == 1 && lons(ind) > 0 % route goes from west to east
        lon_west = lons(1:ind);
        lon_east = lons((ind+1):end);
    elseif length(ind) == 1
        %Route goes from east to west
        lon_west = lons((ind+1):end);
        lon_east = lons(1:ind);
    elseif lons(ind(1)) > 0 % Route crossing from west to east ...
        initially
        lon_west = [lons(1:ind(1)), lons((ind(2)+1):end)];
        lon_east = lons((ind(1)+1):ind(2));
    else
```

```
70          %Route crossing from east to west initially
71          lon_west = lons((ind(1)+1):ind(2));
72          lon_east = [lons(1:ind(1)), lons((ind(2)+1):end)];
73      end
74      min_lon = min(lon_west);
75      max_lon = max(lon_east);
76
77      %Longitudinal span
78      lon_span = (max_lon+360) - min_lon;
79
80      % Origo of orthogonal map
81      if abs(max_lon)>abs(min_lon)
82          lon_origo = min_lon + lon_span/2;
83      else
84          lon_origo = max_lon - lon_span/2;
85      end
86  end
87
88  % Add buffer to map limits
89  lon_buffer = 24;
90  mlon_min = min_lon - lon_buffer/2;
91  mlon_max = max_lon + lon_buffer/2;
92
93  % Set latitude limits for mercantor map
94
95  % Minimum latitude span for visual purposes
96  lat_span_min = (9/16) * (lon_span+lon_buffer);
97  lat_diff = max_lat - min_lat;
98  if lat_diff < lat_span_min
99      extend = max([((lat_span_min - lat_diff)*0.5), 4]);
100 else
101     extend = 4;
102 end
103 mlat_min = max([min_lat - extend, -85]);
104 mlat_max = min([max_lat + extend, 85]);
105
106
107 % Label spacing for mercantor map
108 if lon_span > 300
109     label_space = 45;
110 elseif lon_span > 100
111     label_space = 30;
112 else
113     label_space =15;
114 end
115
116 %% Generate ortholgonal map
117 f1 = figure('color','w');
118 ha_o = axesm('mapproj', 'ortho', 'origin',[lat_origo lon_origo]);
119 axis off, gridm on, framem on;
120 setm(ha_o,'MLineLocation',15,'PLineLocation',15);
121 mlabel on, plabel on;
122 mlabel('equator')
123 plabel(lon_origo-45);
124 plabel('fontweight','bold')
125 setm(gca,'MLabelLocation',30, 'PLabelLocation',15)
126
```

```
% Display geography
geoshow([land.Lat],[land.Lon], 'color', [0 .26 .15])
geoshow([lake.Lat],[lake.Lon],'color',[.24 .41 .99])
geoshow([island.Lat],[island.Lon], 'color', [0 .26 .15])
geoshow([pond.Lat],[pond.Lon],'color',[.24 .41 .91])
geoshow([ice.Lat],[ice.Lon], 'color', [.19 .33 .81])
geoshow([antartic.Lat],[antartic.Lon], 'color', [.1 .1 .1])

% Display cities
geoshow( 'worldcities.shp' , 'Marker' , '.' , 'Color' , 'red' )

%Display ports
geoshow(ports(1,1),ports(1,2),'DisplayType','point',...
'markeredgecolor','k','markerfacecolor','k','marker','o')
textm(ports(1,1),ports(1,2), '  Origin')
geoshow(ports(2,1),ports(2,2),'DisplayType','point',...
'markeredgecolor','k','markerfacecolor','k','marker','o')
textm(ports(2,1),ports(2,2), '  Destination')

% Compute points for great circle route.
gcpts = track2('gc',ports(1,1),ports(1,2),ports(2,1), ports(2,2));
% %Compute waypoints for route divided in n legs of the given ...
    maximum length.
% [latpts, lonpts] = ...
    waypoints(ports(1,1),ports(1,2),ports(2,1),ports(2,2));

% Display great circle route
geoshow(gcpts(:,1),gcpts(:,2),'DisplayType','line',...
    'color','blue','linestyle','--')
% geoshow(latpts,lonpts,'DisplayType','line',...
%     'color',[.4 .2 0],'linestyle','-')

% Display actual route
H1 = geoshow(lats, lons,...
    'DisplayType','line','color','green','linestyle','-');

%% Generate Mercantor map

f2 = figure('color','w');
ha = axesm('mapproj','mercator',...
           'maplatlim',[mlat_min mlat_max],'maplonlim',[mlon_min ...
               mlon_max]);
axis off, gridm on, framem on;
setm(ha,'MLineLocation',15,'PLineLocation',15);
mlabel on, plabel on;
setm(gca,'MLabelLocation',label_space)

% Display geography
geoshow([land.Lat],[land.Lon], 'color', [0 .26 .15])
geoshow([lake.Lat],[lake.Lon],'color',[.24 .41 .99])
geoshow([island.Lat],[island.Lon], 'color', [0 .26 .15])
geoshow([pond.Lat],[pond.Lon],'color',[.24 .41 .91])
geoshow([ice.Lat],[ice.Lon], 'color', [.19 .33 .81])
geoshow([antartic.Lat],[antartic.Lon], 'color', [.1 .1 .1])

% Display cities
geoshow( 'worldcities.shp' , 'Marker' , '.' , 'Color' , 'red' )
```

```
181
182 %Display ports
183 geoshow(ports(1,1),ports(1,2),'DisplayType','point',...
184 'markeredgecolor','k','markerfacecolor','k','marker','o')
185 textm(ports(1,1),ports(1,2), '  Origin')
186 geoshow(ports(2,1),ports(2,2),'DisplayType','point',...
187 'markeredgecolor','k','markerfacecolor','k','marker','o')
188 textm(ports(2,1),ports(2,2), '  Destination')
189
190 % Display great circle route
191 geoshow(gcpts(:,1),gcpts(:,2),'DisplayType','line',...
192     'color','green','linestyle','--')
193 % geoshow(latpts,lonpts,'DisplayType','line',...
194 %     'color',[.4 .2 0],'linestyle','-')
195
196 % Display paths
197 H2 = geoshow(lats, lons,'DisplayType','line', ...
      'color','blue','linestyle','-');
```

D.2 Second level functions

The following functions are called in the function GetRoute.m.

D.2.1 waypoints.m

```
1  function [latpts,lonpts] = waypoints(lat1, lon1, lat2, lon2)
2  % Author:   Ole Brynjar Helland Paulsen
3  % Date:     08.06.2018
4
5  % Function that returns coordinates for the waypoints of a great circle
6  % route divided in n legs, of lenght smaller or equal to the given ...
       maximum
7  % length.
8
9  global max_length;
10
11 % Get gt. circle dist (deg)
12 dgc = distance('gc',lat1, lon1, lat2, lon2);
13
14 % Nautical mi along great circle
15 distgcnm = deg2nm(dgc);
16
17 % Number of sailing legs based on maximum length
18 nlegs = ceil(distgcnm/max_length);
19
20 %Compute waypoints
21 [latpts,lonpts] = gcwaypts(lat1, lon1, lat2, lon2,nlegs);
22 end
```

D.2.2 polycheck.m

```
function [ID] = polycheck(shorelines, latpts, lonpts, startport, ...
    nextport, hull, ports, cross_date)
% Author:   Ole Brynjar Helland Paulsen
% Date:     08.04.2020

% Function for checking if the given route intersects any land ...
    polygons. If
% so, the function returns the specific ID for the land polygon.
% When the route intersect the date line, the route must be split ...
    at the
% date line in order the polyxpoly function to work properly. ...
    Continents
% are selected prior to other polygons. If several continent, the ...
    one also
% intersected by a straight line is chosen. Else the lagerst land areas
% continent or other polygon is chosen.

% Number of shorline polygons
nmax = length(shorelines);
% Variable for counting intersection with land polygons
num_poly = 0;
% Variable for counting intersection with a continent
num_continents = 0;
% Variable for counting number of intersected continents also ...
    intersected
% by a straight line.
num_straight = 0;

% Variables
poly_id = zeros(nmax,1);
poly_area = zeros(nmax,1);
continent_ID = zeros(6,1);
continent_area = zeros(6,1);
straight_ID = zeros(6,1);
straight_span = zeros(6,1);
straight_area = zeros(6,1);

if nmax < 100
    n = 1;
else
    n = min([round(nmax/100), 5]);
end

split = round(nmax/n);

for parts = 1:n
    if n == 1
        lmt_1 = 1;
        lmt_2 = nmax;
    else
        lmt_1 = (parts-1)*split + 1;
        if parts == n
            lmt_2 = nmax;
```

```
48          else
49              lmt_2 = (parts*split);
50          end
51      end
52
53      shore_lats = [shorelines(lmt_1:lmt_2).Y,];
54      shore_lons = [shorelines(lmt_1:lmt_2).X,];
55      [¬, ¬, num_points] = inter_coords(latpts, lonpts, shore_lats,...
56          shore_lons, cross_date);
57
58      if num_points > 0
59          for i = lmt_1:lmt_2
60              %Run through all shorline polygons to check for ...
                    intersections
61
62              %Coordinates and number of unique points of intersection
63              [¬, ¬, num] = inter_coords(latpts, lonpts, ...
                    shorelines(i).Y,...
64                  shorelines(i).X, cross_date);
65
66              % Check if route intersects polygon in more than one point
67              if num > 1
68                  index = find([hull.ID]==shorelines(i).GSHHS_ID);
69                  num_poly = num_poly +1;
70
71                  if shorelines(i).GSHHS_ID > 6
72                      % If polygon is not a continent, the polygon ...
                            area and
73                      % the corresponding ID is stored.
74                      poly_id(num_poly) = shorelines(i).GSHHS_ID;
75                      poly_area(num_poly) = hull(index).Area;
76                  else
77                      num_continents = num_continents + 1;
78
79                      % Check if straight line between the ports also
80                      % intersect the continent.
81                      [inter_y, inter_x, ns] = ...
                            inter_coords(ports(:,1), ...
82                          ports(:,2), shorelines(i).Y, ...
                                shorelines(i).X, ...
83                          cross_date);
84                      if ns > 1
85                          num_straight = num_straight + 1;
86
87                          %Distance between first and last point of
88                          %intersection
89                          [arclen,¬] = distance(inter_y(1),inter_x(1),...
90                              inter_y(end),inter_x(end));
91
92                          straight_ID(num_straight) = ...
                                shorelines(i).GSHHS_ID;
93                          straight_span(num_straight) = deg2nm(arclen);
94                          straight_area(num_straight) = hull(index).Area;
95                      end
96
97                      continent_ID(num_continents) = ...
                            shorelines(i).GSHHS_ID;
```

```
                    continent_area(num_continents) = hull(index).Area;
                end
            end
        end
    end
end

if num_poly == 0
    fprintf('Route between %s and %s does not intersect land.\n\n',...
        startport,nextport);
    ID = [];
elseif num_continents == 0
    %When no continents are intersected, the land polygon with the ...
        largest
    %area is chosen for creating the first path
    area = poly_area(1:num_poly);
    [¬, ind] = max(area);
    ID = poly_id(ind);
    fprintf('Route between %s and %s intersects %d land polygon.\n',...
        startport,nextport, num_poly);
elseif num_continents == 1
    % When one continent is intersected, that polygon is selected.
    ID = continent_ID(1);
    fprintf('Route between %s and %s intersects 1 continent.\n',...
        startport,nextport);
elseif num_straight == 1
    % When several continents are intersected, select the continen ...
        that is
    % also intersected by a straight line, if apliccable.
    ID = straight_ID(1);
    fprintf('Route between %s and %s intersects %d continents.\n',...
        startport,nextport, num_continents)
elseif num_straight > 1
    % Largest area continents
    area = straight_area(1:num_straight);
    straight_ID(1:num_straight);

    % Largest span continents
    span = straight_span(1:num_straight);

    % Index of largest span
    [¬, ind1] = max(span);

    % Sorted span
    span_sort = sort(span);

    % Index of second largest span
    ind2 = find(span == span_sort(end-1));

    % Two gratest spans
    span = [span(ind1), span(ind2)];

    % Area of the same continents
    area = [area(ind1), area(ind2)];

    % Max area continent
    [¬, ind_A] = max(area);
```

```
153
154     % Max span continent
155     [¬, ind_S] = max(span);
156
157     if ind_A == ind_S
158         ID = straight_ID(ind1);
159     elseif span(1) > 2*span(2)
160         ID = straight_ID(ind1);
161     else
162         ID = straight_ID(ind2);
163     end
164     fprintf('Route between %s and %s intersects %d continents.\n',...
165         startport,nextport, num_continents)
166 else
167     % Else select the continent with largest area.
168     area = continent_area(1:num_continents);
169     [¬, ind] = max(area);
170     ID = continent_ID(ind);
171     fprintf('Route between %s and %s intersects %d continents.\n ...
          The largest continent is chosen.\n',startport,nextport, ...
          num_continents)
172 end
173 if num_poly > 0
174     fprintf('Route between %s and %s intersects %d land polygon in ...
          total.\n',startport,nextport, num_poly);
175 end
176 end
```

D.2.3 antarctic.m

```
1  function [connect_nodes] = antarctic(hull_lat, hull_lon, latpts, ...
       lonpts, polar, cross_date)
2  % Author:   Ole Brynjar Helland Paulsen
3  % Date:     01.05.2020
4
5  % This function calculates connection nodes for route crossing the
6  % antarctic continent. The connection nodes are deterimned from the ...
       points
7  % of intersection. One connection node is given for each path, one ...
       on each
8  % side of the polygon. The connection node is given as the middle node
9  % between the two nodes nearest the points of intersection.
10 % If polar sailing is restricted, an alternate modified
11 % hull is used in order to keep the route from going below a given
12 % latitude. Also the shortest path is given two connection nodes in ...
       order
13 % to keep route on path along the latitude limit given.
14
15 %% Find points of intersection
16 [y, x, ¬] = inter_coords(latpts, lonpts, hull_lat, hull_lon, ...
       cross_date);
17
18 %% Find node nearest to the two points of intersection
19 node = zeros(length(x),1);
```

```
num_nodes = size(hull_lat,2)-1;

%Searching for the hull node nearest the point of intersection. The ...
    search
%is limited by only searching the half of the hull nodes nearest in
%longitude.

for i =1:length(x)
    node(i) = nearest_node(hull_lat, hull_lon, y(i), x(i));
end

% Assign connection nodes for each path
if length(unique(node)) == 1
    % Nearest node is identical for both intersection points, set ...
        node as
    % coneection node for path 1
    connect_1 = node(1);
else
    node_lons = hull_lon(unique(node));
    % Identify westward and eastward node
    if max(node_lons)- min(node_lons) > 180
        % Nodes closest to the point of intersection are on the oposite
        % side of the date line
        west_node = find(hull_lon == max(node_lons));
        east_node = find(hull_lon == min(node_lons));
    else
        west_node = find(hull_lon == min(node_lons));
        east_node = find(hull_lon == max(node_lons));
    end

    % Set connection node 1 as middle node between the two hull nodes
    % nearest the two points of intersection
    if west_node > east_node
        %Index of western node is greater than index of eastern node
        mid_node = round(((num_nodes - west_node) + east_node) / 2);
        if (num_nodes - west_node) == east_node
            connect_1 = num_nodes;
        elseif (num_nodes - west_node) > east_node
            connect_1 = west_node + mid_node;
        else
            connect_1 = mid_node - (num_nodes - west_node);
        end
    else
        mid_node = round((east_node - west_node) / 2);
        connect_1 = west_node + mid_node;
    end
end
% Set connect_node 2 as the node oppisite to the first node, as ...
    related to
% node indices, not coordinates.
if connect_1 ≤ num_nodes/2
    connect_2 = connect_1 + round(num_nodes/2);
else
    connect_2 = (connect_1 + round(num_nodes/2))-num_nodes;
end

% Identify the westward port
```

```
74 if cross_date == 1
75     [¬, westport] = max([lonpts(1), lonpts(end)]);
76 else
77     [¬, westport] = min([lonpts(1), lonpts(end)]);
78 end
79
80 %% Assign connection nodes in accordance with the direction of each ...
       path,
81 % and polar sailing option
82 if polar == 1
83     % Sailing in polar waters is allowed. One connection node is ...
          assigned
84     % for each path
85     if westport == 1
86         connect_nodes = [connect_1, connect_1;
87                     connect_2, connect_2];
88     else
89         connect_nodes = [connect_2, connect_2;
90                     connect_1, connect_1];
91     end
92 else
93     % If polar access is resticted, two connect nodes are assigned ...
          for the
94     % shortes path.
95     connect_11 = connect_1 - 1;
96     connect_12 = connect_1 + 1;
97     connect_21 = connect_2 + 1;
98     connect_22 = connect_2 - 1;
99     if connect_1 == 1
100        connect_11 = num_nodes;
101    elseif connect_1 == num_nodes
102        connect_12 = 1;
103    elseif connect_2 == 1
104        connect_22 = num_nodes;
105    elseif connect_2 == num_nodes
106        connect_21 = 1;
107    end
108
109    if westport == 1
110    connect_nodes = [connect_11, connect_12;
111                connect_21, connect_22];
112    else
113    connect_nodes = [connect_22, connect_21;
114                connect_12, connect_11];
115    end
116 end
117 end
```

D.2.4 adjust_ports.m

```
1 function ports_adj = adjust_ports(ports, hull_lon)
2 % Author:   Ole Brynjar Helland Paulsen
3 % Date:     17.06.2018
4
```

```
% This fuction adjusts the sign of the port longitudes, in cases ...
    where the
% ports lie on oppisite side of the international date line, such that
% the logitude coordiates of the ports and the obstacle polygon ...
    have equal
% sign.

% Find the westward and eastward ports.
west_port = find(ports(:,2)>0);
east_port = find(ports(:,2)<0);
fprintf('western port %d\n', west_port)
fprintf('eastern port %d\n', east_port)

% Matrix for the adjusted ports
ports_adj = ports;

%Find the eastward node of the polygon in order to identify if the ...
    polygon
%has positive or negative longitude coordinates.
east_node = max(hull_lon);
% disp(east_node)
if east_node > 0
    disp('Sailing path polygon have positive longitudes')
    % Polygon is located west of the date line, or goes across the date
    % line with longitude coordinates above +180 degrees
    ports_adj(east_port,2) = ports(east_port,2) + 360;
else
    disp('Sailing path polygon have negative longitudes')
    % Polygon is located east of the date line, or it goes across ...
        the date
    % line with longitude coordinates below -180 degrees
    ports_adj(west_port,2) = ports(west_port,2) - 360;
end
end
```

D.2.5 inter_coords.m

```
function [lat, lon, num_points] = inter_coords(latpts, lonpts, ...
    poly_lat, poly_lon, cross_date)
% Author:   Ole Brynjar Helland Paulsen
% Date:     17.06.2018

% This function returns the coordinates of points of intersection ...
    between
% a polygon and a route given by waypoint coordinates.

% The split variable indicates if the ports of the given route has been
% adjusted to equal sign or not. split == 0 indicates that port ...
    coordinates
% have been adjusted so it is not necessary to split the route.
if (max([lonpts(1), lonpts(end)]) > 180 || min([lonpts(1), ...
    lonpts(end)]) < -180)
    split = 0;
else
```

```
    split = 1;
end

if cross_date == 1 && split == 1
    % Split route to check for intersection. If this is not done the
    % polyxpoly function examines a route spanning the globe in opisite
    % direction of the shortest distance across the date line.
    [latpts1, lonpts1, latpts2, lonpts2] = split_route(latpts, lonpts);

    %Points of intersection, if they exists
    [yi1, xi1] = polyxpoly(latpts1, lonpts1, poly_lat, ...
        poly_lon,'unique');
    [yi2, xi2] = polyxpoly(latpts2, lonpts2, poly_lat, ...
        poly_lon,'unique');

    % Coordinates of all points of intersection
    xi = [xi1;xi2];
    yi = [yi1;yi2];

    % Number of intersecting points
    x1 = unique(round(xi1,2),'stable');
    y1 = unique(round(yi1,2),'stable');
    x2 = unique(round(xi2,2),'stable');
    y2 = unique(round(yi2,2),'stable');

    % Coordinates of all points of intersection
    x = [x1;x2];
    y = [y1;y2];
else
    % Points where gc path intersect the shoreline polygon
    [yi, xi] = polyxpoly(latpts, lonpts, poly_lat, poly_lon,'unique');

    %Unique coordinates of point of intersecton
    x = unique(round(xi,2),'stable');
    y = unique(round(yi,2),'stable');
end
% If the number of unique latitude coordinates does not equal the ...
    number of
% unique longitude coordinates, the original intersection ...
    coordinates are
% keept
if length(x) ≠ length(y)
    lat = yi;
    lon = xi;
else
    lat = y;
    lon = x;
end
% Number of unique intersection points
num_points = max([size(x,1),size(y,1)]);

end
```

D.2.6 connecting_outside_non_visi.m

```
function [connect_nodes] = connecting_outside_non_visi(hull_lat, ...
    hull_lon, ports)
% Author:   Ole Brynjar Helland Paulsen
% Date:     19.03.2020

% This function calculates connection nodes for route beteween ...
    ports that
% are both located outside the convex hull, whne the ports are not ...
    visible
% to each other
fprintf('Connection nodes are found from a convex hull\n');

% Create convex hull around nodes made up from the given hull and the
% origin and destination ports. The resulting polygon will go in a
% clockwise maner from port 1 towards port 2, around the modified ...
    hull, and
% back again. All in a clockwise manner.
hlat = [ports(1,1), hull_lat, ports(2,1)];
hlon = [ports(1,2), hull_lon, ports(2,2)];
K = convhull(hlat,hlon);
conv_hull_lat = hlat(K);
conv_hull_lon = hlon(K);

% Divide the convex hull into two seperate paths by finding the element
% number of the destination port. The paths are defined from port 1 to
% port 2. Path 1 goes clockwise around the hull, and path 2 goes ...
    counter
% clockwise. In the case of the two ports having the same latitude,
% the longitude is instead used to identify the correct element.

% Index of destination port
if ports(1,1)==ports(2,1)
    ind = find(conv_hull_lon == ports(2,2));
else
    ind = find(conv_hull_lat == ports(2,1));
end
max_ind = size(conv_hull_lat,2);

% The nodes adjacent to each port is given as connetion nodes
connect_nodes = ...
    [node_index(hull_lat, hull_lon, conv_hull_lat(2), ...
        conv_hull_lon(2)),...
    node_index(hull_lat, hull_lon, conv_hull_lat(ind-1), ...
        conv_hull_lon(ind-1));...
    node_index(hull_lat, hull_lon, conv_hull_lat(max_ind-1), ...
        conv_hull_lon(max_ind-1)),...
    node_index(hull_lat, hull_lon, conv_hull_lat(ind+1), ...
        conv_hull_lon(ind+1))];
end
```

D.2.7 connecting_outside_visi.m

```
function [connect_nodes, dir, inport] = ...
    connecting_outside_visi(hull_lat, hull_lon, ports, latpts, ...
    lonpts, cross_date)
% Author:   Ole Brynjar Helland Paulsen
% Date:     19.03.2020

% This function calculates connection nodes for route beteween ...
    ports that
% are both located outside the convex hull, when the ports are ...
    visible to
% each other. Connection nodes are initally found from points of
% intersection between a great circle route and the sailing path ...
    polygon.
% In som cases this can yeild unfeasible results. Therefore single
% connection nodes from separate convex hulls with each port, are
% investigated. The nodes yeilding shortest path is selected.
fprintf('Both ports are located outside a convex hull around the ...
    current shorline polygon.\n')
fprintf('The ports are visible to each other.\n')

%Pre-allocate variables
shoreline_path(2).Lat = [];
shoreline_path(2).Lon = [];
conv_hull(2).Lat  = [];
inside = zeros(1,2);

% Intersection point between shoreline sailing path and creat circle
% route
[yi, xi, ¬] = inter_coords(latpts, lonpts, hull_lat, hull_lon, ...
    cross_date);

% Nearest node to first and last point of intersection
p1 = nearest_node(hull_lat, hull_lon, yi(1), xi(1));
p2 = nearest_node(hull_lat, hull_lon, yi(end), xi(end));

nodes = [p1, p2];

% Great circle distances from port of origin
arclen1 = distance('gc',[ports(1,1), ports(1,2)],[hull_lat(p1),...
    hull_lon(p1)]);
arclen2 = distance('gc',[ports(1,1), ports(1,2)],[hull_lat(p2),...
    hull_lon(p2)]);

% Find which of the two nodes is nearest the origin port. This node
% will be connection node for origin port. The other node will be
% connection node for the destination port
[¬,ind1] = min([arclen1, arclen2]);

% The other node
A=[1,2];
ind2 = A(A≠ind1);

% Inital connection node for path from intersection points
cn1 = nodes(ind1);
cn2 = nodes(ind2);

p = [1,2];
```

```
51
52  for port = 1:2
53  % Generating convex hull with one port at the time.
54  hlat1 = [ports(port,1), hull_lat];
55  hlon1 = [ports(port,2), hull_lon];
56  K1 = convhull(hlat1,hlon1);
57  conv_hull(port).Lat = hlat1(K1);
58  conv_hull(port).Lon = hlon1(K1);
59
60  % Check if other ports lie inside convex hull
61  other = p(p≠port);
62  [in, on] = inpolygon(ports(other,1), ports(other,2), ...
63      conv_hull(port).Lat, conv_hull(port).Lon);
64  if in == 1 && on == 0
65      inside(other) = 1;
66  end
67
68  end
69
70  if sum(inside) > 0
71      % One of the ports are located within a convex hull made from the
72      % shoreline sailing path and the other port.
73      fprintf('Connection nodes for single path is found.\n')
74      % Identify port inside
75      inport = p(inside == 1);
76
77      connect_nodes = [cn1, cn2];
78
79      dir = 0;
80  else
81      hlat = [ports(1,1), hull_lat, ports(2,1)];
82      hlon = [ports(1,2), hull_lon, ports(2,2)];
83      K = convhull(hlat,hlon);
84      conv_hull_lat = hlat(K);
85      conv_hull_lon = hlon(K);
86
87      % Index of destination port
88      if ports(1,1)==ports(2,1)
89          ind = find(conv_hull_lon == ports(2,2));
90      else
91          ind = find(conv_hull_lat == ports(2,1));
92      end
93      max_ind = size(conv_hull_lat,2);
94
95      % If port 2(ind) is the second node in the convex hull, path 1 ...
            is a
96      % straight line between the ports. If port 2 is the node before the
97      % max node, path 2 is a straight line.
98      path = find([2, (max_ind-1)] == ind);
99
100     %Index of the path that is along the other side of the hull
101     other = find([1,2]≠path);
102
103     node(path).Lat = conv_hull(path).Lat(2);
104     node(path).Lon = conv_hull(path).Lon(2);
105     node(other).Lat = conv_hull(other).Lat(end-1);
106     node(other).Lon = conv_hull(other).Lon(end-1);
```

```
107
108     % Single connection nodes found from seperate convex hulls
109     n3 = node_index(hull_lat, hull_lon, node(1).Lat, node(1).Lon);
110     n4 = node_index(hull_lat, hull_lon, node(2).Lat, node(2).Lon);
111
112     %% Find connection nodes from shortest distance route
113
114     % Find shoreline sailing path
115     if cn1 == cn2
116         shoreline_path(path).Lat = hull_lat(cn1);
117         shoreline_path(path).Lon = hull_lon(cn1);
118     elseif path == 1 && (cn1 < cn2)
119         shoreline_path(path).Lat = hull_lat(cn1:cn2);
120         shoreline_path(path).Lon = hull_lon(cn1:cn2);
121     elseif path == 1
122         shoreline_path(path).Lat = [hull_lat(cn1:end-1), ...
                hull_lat(1:cn2)];
123         shoreline_path(path).Lon = [hull_lon(cn1:end-1), ...
                hull_lon(1:cn2)];
124     elseif path == 2 && (cn1 > cn2)
125         shoreline_path(path).Lat = fliplr(hull_lat(cn2:cn1));
126         shoreline_path(path).Lon = fliplr(hull_lon(cn2:cn1));
127     else
128         shoreline_path(path).Lat = fliplr([hull_lat(cn2:end-1),...
129             hull_lat(1:cn1)]);
130         shoreline_path(path).Lon = fliplr([hull_lon(cn2:end-1),...
131             hull_lon(1:cn1)]);
132     end
133
134     % Distance along shoreline
135     if length(shoreline_path(path).Lat) > 1
136         [¬, shore_dist] = legs(shoreline_path(path).Lat,...
137             shoreline_path(path).Lon,'rh');
138     else
139         shore_dist = 0;
140     end
141
142     % Total distance for route with connection nodes from great circle
143     % intersection
144     dist1 = sum(shore_dist) + deg2nm(distance('gc',ports(1,1),...
145         ports(1,2), hull_lat(cn1), hull_lon(cn1))) + ...
146         deg2nm(distance('gc',hull_lat(cn2), hull_lon(cn2), ...
147         ports(2,1), ports(2,2)));
148
149     % Distance great circle route betweeen ports via nodes found ...
            from convex hull
150     dist2 = deg2nm(distance('gc',ports(1,1), ports(1,2), ...
            hull_lat(n3),...
151         hull_lon(n3))) + deg2nm(distance('gc',hull_lat(n3),...
152         hull_lon(n3), ports(2,1), ports(2,2)));
153     dist3 = deg2nm(distance('gc',ports(1,1), ports(1,2), ...
            hull_lat(n4),...
154         hull_lon(n4))) + deg2nm(distance('gc',hull_lat(n4),...
155         hull_lon(n4), ports(2,1), ports(2,2)));
156
157     % Shorest distance
158     [¬, i_min] = min([dist1, dist2, dist3]);
```

```
159
160       % Possible connection nodes
161       c_nodes = [cn1, cn2;
162                  n3, n3;
163                  n4, n4];
164
165       % Connection nodes from shortest paths
166       connect_nodes(path,:) = c_nodes(i_min,:);
167
168       %  For the other path the indices are kept
169       if other == 1
170           connect_nodes(other,:) = ...
171               [node_index(hull_lat, hull_lon, conv_hull_lat(2),...
172               conv_hull_lon(2)),...
173           node_index(hull_lat, hull_lon, conv_hull_lat(ind-1),...
174           conv_hull_lon(ind-1))];
175       else
176           connect_nodes(other,:) = ...
177               [node_index(hull_lat, hull_lon, ...
                       conv_hull_lat(max_ind-1), ...
                       conv_hull_lon(max_ind-1)),...
178           node_index(hull_lat, hull_lon, conv_hull_lat(ind+1), ...
               conv_hull_lon(ind+1))];
179       end
180       dir=[];
181       inport=[];
182   end
183   end
```

D.2.8 connecting_non_visi_inside.m

```
1   function [connect_nodes, dir, inport] = ...
        connecting_non_visi_inside(hull_lat, hull_lon, port_lats, ...
        port_lons, num_nodes)
2   % Author:   Ole Brynjar Helland Paulsen
3   % Date:     31.03.2020
4
5   % This function calculates connection nodes for route beteween ...
        ports, when
6   % both port are located inside of the convex hull, and the ports are
7   % not visible to each other
8
9   %Both ports are located inside the convex hull
10  fprintf('Both ports are inside the convex hull around the current ...
        land polygon.\n');
11
12  % Variable indicating if port is inside convex hull or sailing path ...
        polygon
13  inport = 1;
14
15  %Find nearest node to the each port
16  nearest = [nearest_node(hull_lat, hull_lon, port_lats(1), ...
        port_lons(1));
17          nearest_node(hull_lat, hull_lon, port_lats(2), port_lons(2))];
```

```
18  if nearest(1) == nearest(2)
19      % The ports share nearest node
20      fprintf('The ports have common nearest node. Single connection ...
            node found.\n');
21      % Nearest node set as connection node for both ports
22      connect_nodes(1,1:2) = nearest(1);
23
24      % Find second nearest node for the origin port
25      nearest2 = nearest_node2(nearest(1), hull_lat, hull_lon, ...
26          port_lats(1), port_lons(1), num_nodes);
27
28      % Set directions for visibility check from indices of two nearest
29      % nodes
30      if (nearest2 == num_nodes && nearest(1) == 1) || nearest(1) > ...
            nearest2
31          dir = [-1, 1];
32      else
33          dir = [1, -1];
34      end
35
36  else
37      % The ports does not share nearest node. Check if the ports have
38      % adjacent nodes in common
39
40      % Find adjacent nodes for each nearest node
41      nodes1 = adjacent_nodes(nearest(1), num_nodes);
42      nodes2 = adjacent_nodes(nearest(2), num_nodes);
43
44      nodes = [nodes1; nodes2];
45
46      if num_nodes > 50
47
48      % For large polygons add two adjacent nodes for each nearest node
49      new_nodes = zeros(2,7);
50      for port = 1:2
51          if nodes(port,end) == num_nodes
52              n2 = 1;
53          else
54              n2 = nodes(port,end) + 1;
55          end
56          if nodes(port,1) == 1
57              n1 = num_nodes;
58          else
59              n1 = nodes(port,1)-1;
60          end
61          new_nodes(port,:) = [n1, nodes(port,:), n2];
62      end
63      nodes = new_nodes;
64      end
65
66      % Check if ports share adjacent nodes
67      [ind1, ¬] = find(nodes1(:) == nodes2);
68      if length(ind1) == 1
69          % Ports have one adjacent node in common, the node is set as
70          % single connection node for both ports
71          connect_nodes(1,1:2) = nodes1(ind1);
```

```
72          fprintf('The ports are located near each other. One ...
                connection node found.\n');
73          % Directions for visibility check
74          if ind1 == 1
75              dir = [1, -1];
76          else
77              dir = [-1, 1];
78          end
79
80      elseif length(ind1) > 1
81          % Ports share several adjacent nodes. Search through common ...
                nodes
82          % to find shortest distance.
83          common_nodes = nodes1(ind1);
84          dist_n = zeros(length(ind1),1);
85          for n = 1:length(ind1)
86              lati = hull_lat(common_nodes(n));
87              long = hull_lon(common_nodes(n));
88          dist_n(n) = ...
89              distance('gc',port_lats(1), port_lons(1), lati, long) ...
90              + distance('gc',lati, long, port_lats(2), port_lons(2));
91          end
92          [¬, min_n] = sort(dist_n);
93          % Common node yielding shortest distance set as single ...
                connection
94          % node
95          connect_nodes(1,1:2) = common_nodes(min_n(1));
96          fprintf('The ports are located near each other. One ...
                connection node found.\n');
97
98          % Directions for visibility check.
99          if ind1(1) == 1
100             dir = [1, -1];
101         else
102             dir = [-1, 1];
103         end
104
105     else
106         % The ports does not share adjacent nodes the furthest adjacent
107         % nodes are set as connection nodes for each port
108         adj_nodes = zeros(2,2);
109
110         if abs(nearest(1) - nearest(2)) > 20 && num_nodes > 50
111
112             % Expand adjacent nodes when nearest nodes are far ...
                    apart and
113             % the sailing path polygon is large
114             new_nodes = zeros(2,15);
115             for p = 1:2
116             n1 = adjacent_nodes(nodes(p,1), num_nodes);
117             n11 = adjacent_nodes(n1(1), num_nodes);
118             n2 = adjacent_nodes(nodes(p,end), num_nodes);
119             n22 = adjacent_nodes(n2(end), num_nodes);
120
121             new_nodes(p,:) = [n11(1:4),nodes(p,:), n22(2:5)];
122             end
123
```

```
124             nodes = new_nodes;
125         end
126
127         for port  = 1:2
128             adj_nodes(:,port) = [nodes(port,end);
129                                  nodes(port,1)];
130         end
131         connect_nodes = [adj_nodes(1,1), adj_nodes(2,2);
132                          adj_nodes(2,1), adj_nodes(1,2)];
133         fprintf('Four connection nodes are found.\n');
134         dir =[];
135         inport =[];
136     end
137 end
138 end
```

D.2.9 connecting_non_visi_in_out.m

```
1  function [connect_nodes, dir, inport] = ...
       connecting_non_visi_in_out(hull_lat, hull_lon, port_lats, ...
       port_lons, num_nodes, in)
2  % Author:   Ole Brynjar Helland Paulsen
3  % Date:     31.03.2020
4
5  % This function calculates connection nodes for route beteween ...
       ports, when
6  % one port is located inside of the convex hull, and the ports are
7  % not visible to each other
8
9  % Indentify port inside convex hull
10 [¬,inport] = max(in);
11 [¬,outport] = min(in);
12
13 fprintf('Port %d is inside the convex hull of the given land ...
       polygon.\n',inport);
14
15 % Ports are not visible to each other. The ports can still be on
16 % the same side of the land polygon with a small area intersected by
17 % the route. In such cases the connection node from a convex hull
18 % will be beyond the nodes nearest the inport.
19
20 % Generating convex hull with the outside port to find the proper
21 % connection nodes
22 hlat1 = [port_lats(outport), hull_lat];
23 hlon1 = [port_lons(outport), hull_lon];
24 K1 = convhull(hlat1,hlon1);
25 conv_h_lat = hlat1(K1);
26 conv_h_lon = hlon1(K1);
27 ind_max = size(conv_h_lat,2);
28
29 % Connection nodes for the outside port
30 A = node_index(hull_lat, hull_lon, conv_h_lat(2), conv_h_lon(2));
31 C = node_index(hull_lat, hull_lon, conv_h_lat(ind_max-1), ...
       conv_h_lon(ind_max-1));
32
```

```
% disp('connection nodes for outport:A,C')
% disp(A)
% disp(C)


%Find hull node nearest the port located inside the convex hull
nearest = nearest_node(hull_lat, hull_lon, port_lats(inport), ...
    port_lons(inport));
% Find the adjacent nodes
nodes = adjacent_nodes(nearest, num_nodes);

% Set the adjacent node in each direction as connect node for the
% port inside the convex hull
% disp('nearest node to inport')
% disp(nearest)
if num_nodes < 50
    B = nodes(1);
    D = nodes(end);
else
    if nodes(end) == num_nodes
        D = 1;
    else
        D = nodes(end) + 1;
    end
    if nodes(1) == 1
        B = num_nodes;
    else
        B = nodes(1) - 1;
    end
%     disp('connection nodes for inport:B,D')
%     disp(B)
%     disp(D)
end

% Check if connection node from convex hull is located beyond the
% connection nodes for the port inside the hull.
if A == D
    fprintf('The ports are located on the same side of land area. ...
        One connection node found.\n');
    connect_nodes(1,1:2) = B;
    dir = 0;
elseif B == C
    fprintf('The ports are located on the same side of land area. ...
        One connection node found.\n');
    connect_nodes(1,1:2) = D;
    dir = 0;
elseif (B>C && A>B) || (C>A && A>B) || (C>A && B>C) || (D>C && B>D) ...
    || (C>B && D>C) || (C>B && B>D)
    fprintf('The ports are located on the same side of land area. ...
        One connection node found.\n');
    % Only 5 nearest nodes
    if length(nodes) == 7
        nodes(end) = [];
        nodes(1) = [];
    end
    % Find node yeilding shortest distance
    dist_n = zeros(length(nodes),1);
```

```
86
87      for n = 1:length(nodes)
88          lati = hull_lat(nodes(n));
89          long = hull_lon(nodes(n));
90      dist_n(n) = ...
91          distance('gc',port_lats(inport), port_lons(inport), lati, ...
                long)...
92          + distance('gc',lati, long, port_lats(outport), ...
93          port_lons(outport));
94      end
95      [¬, min_n] = sort(dist_n);
96      connect_nodes(1,1:2) = nodes(min_n(1));
97      dir = 0;
98  else
99      fprintf('The ports lie on opposite sides of the land area.\n');
100     % Ports lie on oppisite side of land area. Set connection nodes
101     % from the estabilised values
102     connect_nodes = zeros(2,2);
103     connect_nodes(outport,outport) = A;
104     connect_nodes(outport,inport) = B;
105     connect_nodes(inport,outport) = C;
106     connect_nodes(inport,inport) = D;
107     dir = [];
108 end
109 end
```

D.2.10 connecting_visi.m

```
1   function [connect_nodes, dir, inport] = connecting_visi(hull_lat, ...
        hull_lon, port_lats, port_lons, num_nodes, latpts, lonpts, ...
        cross_date, in_conv, in, on)
2   % Author:   Ole Brynjar Helland Paulsen
3   % Date:     31.03.2020
4
5   % This function calculates connection nodes for route beteween ...
        ports, where
6   % one or both ports lie inside a convex hull around a land area, ...
        and the
7   % ports are visible to each other
8   fprintf('The ports are visible to each other.\n');
9
10  % Index of ports
11  port_ind = [1,2];
12
13  % Varible indication direction for adjusting ports during if not ...
        visible
14  % to each other. If both ports are inside the convex hull, or both ...
        inside
15  % the modified hull, the direction is established here. If not the
16  % direction is established in a seperate function.
17  dir = 0;
18
19  % Variable indication which port is inside a convex hull or sailing ...
        path
```

```
% polygon. Randomly set origin port as inport
inport = 1;

% Number of ports inside or on border of modified hull.
in_mod = sum(in);

% Number of ports on border of modified hull
on_mod = sum(on);

if on_mod == 2
    %Both ports are located on border of modified hull
    fprintf('Both ports are on border of modified hull, 2 ...
        connection nodes found.\n');
    connect_nodes = zeros(2);

    for port = 1:2
        Lat = port_lats(port);
        Lon = port_lons(port);

        % Hull node nearest port
        nearest1 = nearest_node(hull_lat, hull_lon, Lat, Lon);

        % Second nearest node
        nearest2 = nearest_node2(nearest1, hull_lat, hull_lon, Lat,...
            Lon, num_nodes);

        %Indices of two nearest nodes
        nodes = [nearest1, nearest2];

        % Other port than current
        other_port = port_ind(port_ind≠port);

        % Coordinares of other port
        Lat2 = port_lats(other_port);
        Lon2 = port_lons(other_port);

        % Distance from other port to the two nodes nearest current ...
            port
        d3 = distance('rh',[Lat2,Lon2],[hull_lat(nearest1),...
            hull_lon(nearest1)]);
        d4 = distance('rh',[Lat2,Lon2],[hull_lat(nearest2),...
            hull_lon(nearest2)]);

        % Choose nearest node yielding the shortest distance
        [¬, n1] = min([d3, d4]);

        % Adjacent node nearest to other port set as connection node;
        connect_nodes(port) = nodes(n1);

    end

    % Since both ports are located on border of sailing path ...
        polygon, and
    % the nearest node is set as connection node, visibility is ...
        ensured.
    dir = [];

```

```
73  elseif in_mod == 2 || (in_mod == 0 && sum(in_conv) == 2)
74      % Both ports either inside modified hull, or inside convex ...
            hull, but
75      % outside modified hull
76
77      %Find nearest node to the each port
78      nearest = [nearest_node(hull_lat, hull_lon, port_lats(1), ...
79          port_lons(1));
80              nearest_node(hull_lat, hull_lon, port_lats(2), ...
                    port_lons(2))];
81
82      if nearest(1) == nearest(2)
83          % The ports share nearest node
84
85          % Nearest node set as connection node for both ports
86          connect_nodes(1,1:2) = nearest(1);
87
88          % Find second nearest node for the origin port
89          nearest2 = [nearest_node2(nearest(1), hull_lat, hull_lon, ...
90              port_lats(1), port_lons(1), num_nodes);
91                      nearest_node2(nearest(1), hull_lat, hull_lon, ...
92                      port_lats(2), port_lons(2), num_nodes)];
93          if nearest2(1) == nearest(2)
94              % Both nearest nodes in common, visibility is assumed ...
                    from near
95              % proximity
96              dir = [];
97          elseif (nearest2(1) == num_nodes && nearest(1) == 1) || ...
                nearest(1) > nearest2(1)
98              % Set directions for visibility check from indices of two
99              % nearest nodes.
100             dir = [-1, 1];
101         else
102             dir = [1, -1];
103         end
104
105     else
106         % The ports does not share nearest node.
107
108         % Find adjacent nodes for each nearest node
109         nodes1 = adjacent_nodes(nearest(1), num_nodes);
110         nodes2 = adjacent_nodes(nearest(2), num_nodes);
111
112         % Check if the ports have adjacent nodes in common
113         [ind1, ¬] = find(nodes1(:) == nodes2);
114         if length(ind1) == 1
115             % Ports have one adjacent node in common, the node is ...
                    set as
116             % single connection node for both ports
117             connect_nodes(1,1:2) = nodes1(ind1);
118             fprintf('The ports are located near each other. One ...
                    connection node found.\n');
119             % Directions for visibility check
120             if ind1 == 1
121                 dir = [1, -1];
122             else
123                 dir = [-1, 1];
```

```
124             end
125
126         elseif length(ind1) > 1
127             % Ports share several adjacent nodes. Search through common
128             % nodes to find shortest distance.
129             common_nodes = nodes1(ind1);
130             dist_n = zeros(length(ind1),1);
131             for n = 1:length(ind1)
132                 lati = hull_lat(common_nodes(n));
133                 long = hull_lon(common_nodes(n));
134             dist_n(n) = ...
135                 distance('gc',port_lats(1), port_lons(1), lati, ...
                        long) ...
136                 + distance('gc',lati, long, port_lats(2), ...
                        port_lons(2));
137             end
138             [¬, min_n] = sort(dist_n);
139             % Common node yielding shortest distance set as single
140             % connection node
141             connect_nodes(1,1:2) = common_nodes(min_n(1));
142             fprintf('The ports are located near each other. One ...
                    connection node found.\n');
143
144             % Directions for visibility check.
145             if ind1(1) == 1
146                 dir = [1, -1];
147             else
148                 dir = [-1, 1];
149             end
150         else
151             % If ports does not share adjacent nodes the furthest ...
                    adjacent
152             % nodes are set as inital connection nodes for each port
153
154             connect_init = [nodes1(end),nodes2(1);
155                             nodes1(1),nodes2(end)];
156
157
158             % Distance between initial connection nodes for each path
159             d1 = distance('rh',[hull_lat(connect_init(1,1)), ...
160                 hull_lon(connect_init(1,1))],...
161                 [hull_lat(connect_init(1,2)), ...
                        hull_lon(connect_init(1,2))]);
162             d2 = distance('rh',[hull_lat(connect_init(2,1)), ...
163                 hull_lon(connect_init(2,1))],...
164                 [hull_lat(connect_init(2,2)), ...
                        hull_lon(connect_init(2,2))]);
165
166             % Find nodes yielding shortest path
167             [¬, n1] = min([d1, d2]);
168
169             connect_nodes =[connect_init(n1,1), connect_init(n1,2)];
170             fprintf('Two connection nodes are found.\n');
171
172             if n1 == 1
173                 dir = [-1, 1];
174             else
```

```
                dir = [1, -1];
            end
        end
    end

elseif on_mod == 1
    % One port on border of sailing path polygon and the other is ...
        outside.

    % Port on border
    inport = port_ind(on);

    fprintf('Port %d is located on the border of the modified ...
        hull.\n Single connection node found from nearest ...
        nodes\n',inport);

    % Hull node nearest port
    nearest1 = nearest_node(hull_lat, hull_lon, port_lats(inport), ...
        port_lons(inport));

    % Second nearest node
    nearest2 = nearest_node2(nearest1, hull_lat, hull_lon, ...
        port_lats(inport), port_lons(inport), num_nodes);

    % Two nearest hull nodes to current port
    nodes = [nearest1, nearest2];

    % Ditance between ports with two different nodes as waypoints
    dist_n = zeros(2,1);
    for n = 1:2
        lati = hull_lat(nodes(n));
        long = hull_lon(nodes(n));
    dist_n(n) = ...
        distance('gc',port_lats(1), port_lons(1), lati, long) ...
        + distance('gc',lati, long, port_lats(2), port_lons(2));
    end

    % Choose nearest node yielding the shortest distance
    [¬, n1] = min(dist_n);

    % Set single connection node
    connect_nodes(1,1:2) = nodes(n1);

elseif in_mod == 1
    % One port inside modified hull and one outside. The latter can be
    % either inside or outside the convex hull

    % Port inside modified hull
    inport = port_ind(in);

    fprintf('Port %d is located inside modified hull. Single ...
        connection node found\n',inport);

    % Point of intersection between straight line and modified hull
    [lat1, lon1, ¬] = inter_coords(port_lats, port_lons, hull_lat, ...
        hull_lon, cross_date);

```

```
228     % Hull node nearest the point of intersection
229     nearest = nearest_node(hull_lat, hull_lon, lat1, lon1);
230     nodes = adjacent_nodes(nearest, num_nodes);
231     num_n = length(nodes);
232     dist_n = zeros(num_n,1);
233
234     for n = 1:num_n
235         lati = hull_lat(nodes(n));
236         long = hull_lon(nodes(n));
237     dist_n(n) = ...
238         distance('gc',port_lats(1), port_lons(1), lati, long) ...
239         + distance('gc',lati, long, port_lats(2), port_lons(2));
240     end
241     [¬, min_n] = sort(dist_n);
242
243     % Single connection node
244     connect_nodes(1, 1:2) = nodes(min_n(1));
245
246 elseif sum(in_conv) == 1
247     % One port is located inside a convex hull around the land ...
            area, but
248     % outside the modified hull. The other port lies outside the convex
249     % hull
250
251     % Identify inside port and outside port
252     [¬,inport] = max(in_conv);
253
254     fprintf('Port %d is located inside a convex hull around the ...
            shorline polygon.\n Single connection node found\n',inport);
255
256     % Find nearest point of intersetion between gc route and ...
            modified hull
257     [yi, xi, ¬] = inter_coords(latpts, lonpts, hull_lat, hull_lon, ...
258         cross_date);
259
260     % Nearest node to first and last point of intersection. It is ...
            assumed
261     % that the first and last intersection point are the two points
262     % nearest each port.
263     p1 = nearest_node(hull_lat, hull_lon, yi(1), xi(1));
264     p2 = nearest_node(hull_lat, hull_lon, yi(end), xi(end));
265
266     nodes = [p1, p2];
267
268     % Great circle distances from port of origin
269     arclen1 = distance('gc',[port_lats(1), port_lons(1)],...
270         [hull_lat(p1), hull_lon(p1)]);
271     arclen2 = distance('gc',[port_lats(2), port_lons(2)],...
272         [hull_lat(p2), hull_lon(p2)]);
273
274     % Find which of the two nodes is nearest the origin port. This node
275     % will be connection node for origin port. The other node will be
276     % connection node for the destination port
277     [¬,ind1] = min([arclen1, arclen2]);
278
279     % The orther node
280     A=[1,2];
```

```
281         ind2 = A(A≠ind1);
282
283         % Connection nodes
284         connect_nodes(1, 1:2) = [nodes(ind1), nodes(ind2)];
285     end
286
287     end
```

D.2.11 one_path.m

```
1   function [path, min_path, max_path, connect_nodes] = ...
        one_path(ports, hull_lat, hull_lon, shore_lat, shore_lon, ...
        connect_nodes, num_nodes, inport, port_node, dir)
2   % Author:   Ole Brynjar Helland Paulsen
3   % Date:     08.05.2020
4
5   % This function generates one path between two points around a modified
6   % hull(polygon used to describe navagable landes around a ...
        shorline), for
7   % instances where a great circle route intersects a land area, but a
8   % straight line between the ports does not. The path consist of three
9   % parts; a route from port A to a connection node, a route along the
10  % hull, and a route from a connecion node to port B.
11
12  %Allocate variables for later use
13  path(2).Lat = [];
14  coord(2).Lat = [];
15
16  %% Find paths from ports to connection nodes
17  if dir == 0
18      dir = direction_visi_check(ports, connect_nodes, num_nodes, ...
19          hull_lat, hull_lon, inport);
20  end
21
22   %Find index of current connection node in the hull polygon
23  for port =1:2
24      % If inport , the port node is set as a
25      % connection node
26      if port_node(port) == 1
27          % Port node is a node in the modified hull
28
29          % Connection node set as the given hull node.
30          connect_nodes(port) = node_index(hull_lat, hull_lon, ...
31              ports(port,1),ports(port,2));
32
33          % Path from port to connection node is set as empty
34          coord(port).Lat = [];
35          coord(port).Lon = [];
36
37      else
38          %Find index of current connection node in the hull polygon
39          i = connect_nodes(port);
40
41          %Route from current point towards index node
```

```
42          [latpts,lonpts] = waypoints(ports(port,1),ports(port,2), ...
43              hull_lat(i),hull_lon(i));
44
45          if isempty(dir) == 1
46              % Visibility is assumed
47              coord(port).Lat = transpose(latpts);
48              coord(port).Lon = transpose(lonpts);
49              dir = direction_visi_check(ports, connect_nodes, ...
50                  num_nodes, hull_lat, hull_lon, inport);
51          else
52              %For each of the connection nodes, check for visibility.
53
54              % Check if route between connection node and port ...
                    crosses the
55              % date line. If longitudenal distance between the port and
56              % connection node spans more than 180 degrees, i.e. the ...
                    path
57              % crosses the dateline. The gcwaypts function will
58              % automatically change sign of the nodes when the path ...
                    crosses
59              % the date line, even when the longitude of the polygon ...
                    node
60              % has equal sign as the port, eg. +-180 degrees. The ...
                    polyxpoly
61              % function requires that both polygons have coordinates of
62              % equal sign.
63              a = find(diff(lonpts>0)≠0);
64              b = abs(lonpts(a)) + abs(lonpts(a+1));
65              num_lonpts = size(lonpts,1);
66
67              if b > 180
68                  if lonpts(a) > 0 && hull_lon(i) > 0
69                  % Route is crossing the date line from west to ...
                        east, as
70                  % do the hull polygon (lon = +180)
71                      lonpts(a+1:num_lonpts) = lonpts(a+1:num_lonpts) ...
                            + 360;
72                  elseif lonpts(a) > 0
73                      % Route goes west to east, and hull longitude is
74                      % negative
75                      lonpts(1:a) = lonpts(1:a) - 360;
76                  elseif lonpts(a) < 0 && hull_lon(i) < 0
77                      % Route goes east to west and hull longitude is
78                      % negative
79                      lonpts(a+1:num_lonpts) = lonpts(a+1:num_lonpts) ...
                            - 360;
80                  else
81                      % Route goes east to west and hull longitude is
82                      % positive
83                  lonpts(1:a) = lonpts(1:a) + 360;
84                  end
85              end
86
87              % If polyxpoly is empty the route will intersect a part ...
                    of the
88              %polygon. If there is intersection the algorithm will ...
                    check the
```

```
89              %neighbouring node in direction towards the current port.
90              [x, y] = polyxpoly(latpts,lonpts,shore_lat,shore_lon);
91              x = unique(round(x,2));
92              y = unique(round(y,2));
93              inter_points = max([size(x,1),size(y,1)]);
94
95              while inter_points > 1
96                  if dir(port) == 1 && i == num_nodes
97                      i =1;
98                  elseif dir(port) == -1 && i == 1
99                      i = num_nodes;
100                 else
101                     i = i + 1*dir(port);
102                 end
103
104                 %Route from current point towards index node
105                 [latpts,lonpts] = ...
                        waypoints(ports(port,1),ports(port,2),...
106                     hull_lat(i),hull_lon(i));
107                 a = find(diff(lonpts>0)≠0);
108                 b = abs(lonpts(a)) + abs(lonpts(a+1));
109                 num_lonpts = size(lonpts,1);
110                 if b > 180
111                     if lonpts(a) > 0 && hull_lon(i) > 0
112                     % Route is crossing the date line from west to ...
                            east,
113                     % as do the hull polygon (lon = +180)
114                         lonpts(a+1:num_lonpts) = ...
                                lonpts(a+1:num_lonpts)...
115                             + 360;
116                     elseif lonpts(a) > 0
117                         %Route goes west to east, negative hull ...
                                longitude
118                         lonpts(1:a) = lonpts(1:a) - 360;
119                     elseif lonpts(a) < 0 && hull_lon(i) < 0
120                         %Route goes east to west, negative hull ...
                                longitude
121                         lonpts(a+1:num_lonpts) = ...
                                lonpts(a+1:num_lonpts)...
122                             - 360;
123                     else
124                         %Route goes east to west, positive hull ...
                                longitude
125                         lonpts(1:a) = lonpts(1:a) + 360;
126                     end
127                 end
128                 [x, y] = polyxpoly(latpts,lonpts,shore_lat,shore_lon);
129                 x = unique(round(x,2));
130                 y = unique(round(y,2));
131                 inter_points = max([size(x,1),size(y,1)]);
132             end
133
134             connect_nodes(port) = i;
135
136             %Route from current point towards index node
137             [latpts,lonpts] = ...
                    waypoints(ports(port,1),ports(port,2), ...
```

```
                hull_lat(i),hull_lon(i));
            coord(port).Lat = transpose(latpts);
            coord(port).Lon = transpose(lonpts);
        end
    end
end

if isempty(coord(1)) == true
    % Port1 is a connection node. Path from port to connection node ...
        empty
    lats1 = [];
    lons1 = [];
else
    n1 = size(coord(1).Lat,2);

    if n1 > 1
        lats1 = coord(1).Lat(1:n1-1);
        lons1 = coord(1).Lon(1:n1-1);
    else
        lats1 = coord(1).Lat;
        lons1 = coord(1).Lon;
    end
end

if isempty(coord(2)) == true
    % Port2 is a hull node. Empty path
    lats2 = [];
    lons2 = [];
else
    n2 = size(coord(2).Lat,2);
    lats_dest = fliplr(coord(2).Lat);
    lons_dest = fliplr(coord(2).Lon);
    if n2 > 1
        lats2 = lats_dest(2:n2);
        lons2 = lons_dest(2:n2);
    else
        lats2 = lats_dest;
        lons2 = lons_dest;
    end
end

%% Extact path along hull, according to the index number of the start
% and end nodes.
start_node = connect_nodes(1);
end_node = connect_nodes(2);

if start_node == end_node
    hull_path_lat = hull_lat(start_node);
    hull_path_lon = hull_lon(start_node);
elseif dir(inport) == 1 && inport == 2  && (start_node < end_node)
    %The hull index number will increase along the path towards the ...
        inport
    %when dir(inport) == 1
    hull_path_lat = hull_lat(start_node:end_node);
    hull_path_lon = hull_lon(start_node:end_node);
elseif dir(inport) == 1  && inport == 2
```

```
192      % If the index number of the end connection node is smaller ...
             than the
193      % the index of the start connection node, the path will pass the
194      % max index node and the next node is node with index one.
195      hull_path_lat = [hull_lat(start_node:num_nodes), ...
             hull_lat(1:end_node)];
196      hull_path_lon = [hull_lon(start_node:num_nodes), ...
             hull_lon(1:end_node)];
197  elseif dir(inport) == 1 && inport == 1  && (start_node > end_node)
198      hull_path_lat = fliplr(hull_lat(end_node:start_node));
199      hull_path_lon = fliplr(hull_lon(end_node:start_node));
200  elseif dir(inport) == 1 && inport == 1
201      % If the index number of the end connection node is larger than the
202      % the index of the start connection node, the path will pass the
203      % max index node and the next node is node with index one.
204      hull_path_lat = fliplr([hull_lat(end_node:num_nodes), ...
205          hull_lat(1:start_node)]);
206      hull_path_lon = fliplr([hull_lon(end_node:num_nodes), ...
207          hull_lon(1:start_node)]);
208  elseif dir(inport) == -1 && inport == 1  && (start_node < end_node)
209      %The hull index number will decrease along the path towards the ...
             inport
210      %when dir(inport) == -1
211      hull_path_lat = hull_lat(start_node:end_node);
212      hull_path_lon = hull_lon(start_node:end_node);
213  elseif dir(inport) == -1  && inport == 1
214      % If the index number of the end connection node is smaller ...
             than the
215      % the index of the start connection node, the path will pass the
216      % max index node and the next node is node with index one.
217      hull_path_lat = [hull_lat(start_node:num_nodes), ...
             hull_lat(1:end_node)];
218      hull_path_lon = [hull_lon(start_node:num_nodes), ...
             hull_lon(1:end_node)];
219  elseif dir(inport) == -1 && inport == 2  && (start_node > end_node)
220      hull_path_lat = fliplr(hull_lat(end_node:start_node));
221      hull_path_lon = fliplr(hull_lon(end_node:start_node));
222  elseif dir(inport) == -1 && inport == 2
223      % If the index number of the end connection node is larger than the
224      % the index of the start connection node, the path will pass the
225      % max index node and the next node is node with index one.
226      hull_path_lat = fliplr([hull_lat(end_node:num_nodes), ...
227          hull_lat(1:start_node)]);
228      hull_path_lon = fliplr([hull_lon(end_node:num_nodes), ...
229          hull_lon(1:start_node)]);
230  end
231
232  %% Combine three parts to complete path
233
234  % The path is is made up from the path from the origin port to the
235  % first connection node, the path along the hull, and the path from the
236  % end connection node to the destination
237  path(1).Lat = [lats1, hull_path_lat, lats2];
238  path(1).Lon = [lons1, hull_path_lon, lons2];
239
240  %Generate consistency such that lognitudes are between [-180, 180]
241  %Find nodes with longitudes above 180 degrees, and convert to negative
```

```
242 %values
243 above = find(path(1).Lon > 180);
244 path(1).Lon(above) = path(1).Lon(above) - 360;
245
246 below = find(path(1).Lon < -180);
247 path(1).Lon(below) = path(1).Lon(below) + 360;
248
249 path(2)= path(1);
250
251 min_path = 1;
252 max_path = 2;
253
254 end
```

D.2.12 two_paths.m

```
1  function [path, min_path, max_path, connect_nodes] = ...
       two_paths(ports, hull_lat, hull_lon, shore_lat, shore_lon, ...
       connect_nodes, num_nodes, polar, port_node)
2  % Author:   Ole Brynjar Helland Paulsen
3  % Date:     08.06.2018
4
5  % This function generates two paths between two points around a ...
       modified
6  % hull(polygon used to describe navagable landes around a ...
       shorline). Each
7  % path consist of three parts; a route from port A to a connection ...
       node, a
8  % route along the hull, and a route from a connecion node to port B.
9
10 %Allocate variables for later use
11 hull_path(2).Lat = [];
12 coord(2).Lat = [];
13 path(2).Lat =[];
14
15 for path_num = 1:2
16
17     for port =1:2
18         if port_node(port) == 1
19             % Port node is node in modified hull.
20
21             % Connection node set as the given hull node.
22             connect_nodes(path_num,port) = node_index(hull_lat, ...
23                 hull_lon, ports(port,1),ports(port,2));
24
25             % Path from port to connection node is set as empty
26             coord(port).Lat = [];
27             coord(port).Lon = [];
28         else
29             %For each of the connection nodes, check for visibility.
30
31             %Find index of current connection node in the hull polygon
32             i = connect_nodes(path_num,port);
33
```

```
34              %Route from current point towards index node
35              [latpts,lonpts] = ...
                    waypoints(ports(port,1),ports(port,2), ...
36                  hull_lat(i),hull_lon(i));
37              % Check if route between connection node and port ...
                    crosses the
38              % date line. If longitudenal distance between the port and
39              % connection node spans more than 180 degrees, i.e. the ...
                    path
40              % crosses the dateline. The gcwaypts function will
41              % automatically change sign of the nodes when the path ...
                    crosses
42              % the date line, even when the longitude of the polygon ...
                    node
43              % has egual sign as the port, eg. +-180 degrees. The ...
                    polyxpoly
44              % function requires that both polygons have coordinates of
45              % equal sign.
46
47              a = find(diff(lonpts>0)≠0);
48              b = abs(lonpts(a)) + abs(lonpts(a+1));
49              num_lonpts = size(lonpts,1);
50              if b > 180
51                  if lonpts(a) > 0 && hull_lon(i) > 0
52                  % Route is crossing the date line from west to ...
                        east, as
53                  % do the hull polygon (lon = +180)
54                      lonpts(a+1:num_lonpts) = lonpts(a+1:num_lonpts) ...
                            + 360;
55                  elseif lonpts(a) > 0
56                      % Route goes west to east, and hull longitude is
57                      % negative
58                      lonpts(1:a) = lonpts(1:a) - 360;
59                  elseif lonpts(a) < 0 && hull_lon(i) < 0
60                      % Route goes east to west and hull longitude is
61                      % negative
62                      lonpts(a+1:num_lonpts) = lonpts(a+1:num_lonpts) ...
                            - 360;
63                  else
64                      % Route goes east to west and hull longitude is
65                      % positive
66                  lonpts(1:a) = lonpts(1:a) + 360;
67                  end
68              end
69
70              % If polyxpoly is empty the route will intersect a part ...
                    of the
71              % polygon. If there is intersection the algorithm will ...
                    check
72              % the neighbouring node in direction towards the ...
                    current port.
73              [x, y] = polyxpoly(latpts,lonpts,shore_lat,shore_lon);
74              x = unique(round(x,2));
75              y = unique(round(y,2));
76              inter_points = max([size(x,1),size(y,1)]);
77
78              % Time limit for while loop is set to 20 seconds
```

```
79              time0 = tic;
80              timeLimit = 20;
81              while inter_points > 1 && toc(time0)<timeLimit
82
83                  % Path 1 will always have a clockwise direction, ...
                        while the
84                  % opposite is true for path 2. The node index of ...
                        path 1
85                  % will therefore increase along the direction of ...
                        travel,
86                  % while the opposite yields true for path 2.
87                  if i == 1 && path_num == port
88                      i = num_nodes;
89                  elseif path_num == port
90                      i=i-1;
91                  elseif i == num_nodes && path_num ≠ port
92                      i = 1;
93                  else
94                      i=i+1;
95                  end
96
97                  %Route from current point towards index node
98                  [latpts,lonpts] = waypoints(ports(port,1),...
99                      ports(port,2), hull_lat(i),hull_lon(i));
100                 a = find(diff(lonpts>0)≠0);
101                 b = abs(lonpts(a)) + abs(lonpts(a+1));
102                 num_lonpts = size(lonpts,1);
103                 if b > 180
104                     if lonpts(a) > 0 && hull_lon(i) > 0
105                     % Route is crossing the date line from west to ...
                            east, as
106                     % do the hull polygon (lon = +180)
107                         lonpts(a+1:num_lonpts) = ...
                                lonpts(a+1:num_lonpts)...
108                             + 360;
109                     elseif lonpts(a) > 0
110                         % Route goes west to east, and hull ...
                                longitude is
111                         % negative
112                         lonpts(1:a) = lonpts(1:a) - 360;
113                     elseif lonpts(a) < 0 && hull_lon(i) < 0
114                         % Route goes east to west and hull ...
                                longitude is
115                         % negative
116                         lonpts(a+1:num_lonpts) = ...
                                lonpts(a+1:num_lonpts)...
117                             - 360;
118                     else
119                         % Route goes east to west and hull ...
                                longitude is
120                         % positive
121                     lonpts(1:a) = lonpts(1:a) + 360;
122                     end
123                 end
124                 [x, y] = polyxpoly(latpts,lonpts,shore_lat,shore_lon);
125                 x = unique(round(x,2));
126                 y = unique(round(y,2));
```

```
127                 inter_points = max([size(x,1),size(y,1)]);
128             end
129
130             connect_nodes(path_num,port) = i;
131
132             %Route from current point towards index node
133             [latpts,lonpts] = waypoints(ports(port,1),ports(port,2),...
134                 hull_lat(i),hull_lon(i));
135             coord(port).Lat = transpose(latpts);
136             coord(port).Lon = transpose(lonpts);
137         end
138     end
139
140
141     % Extact path along hull, according to the index number of the ...
            start
142     % and end nodes.
143     start_node = connect_nodes(path_num,1);
144     end_node = connect_nodes(path_num,2);
145
146     if start_node == end_node
147         hull_path(path_num).Lat = hull_lat(start_node);
148         hull_path(path_num).Lon = hull_lon(start_node);
149     elseif path_num == 1 && (start_node < end_node)
150         %For path 1, the index number will increase along the path ...
                towards
151         %the destination.
152         hull_path(path_num).Lat = hull_lat(start_node:end_node);
153         hull_path(path_num).Lon = hull_lon(start_node:end_node);
154     elseif path_num == 1
155         % If the index number of the end connection node is smaller ...
                than
156         % the index of the start connection node, the path will ...
                pass the
157         % max index node and the next node is node with index one.
158         hull_path(path_num).Lat = [hull_lat(start_node:num_nodes),...
159             hull_lat(1:end_node)];
160         hull_path(path_num).Lon = [hull_lon(start_node:num_nodes),...
161             hull_lon(1:end_node)];
162     elseif path_num == 2 && (end_node < start_node)
163         %For path 2, the index number will decrease along the path ...
                towards
164         %the destination.
165         hull_path(path_num).Lat = ...
                fliplr(hull_lat(end_node:start_node));
166         hull_path(path_num).Lon = ...
                fliplr(hull_lon(end_node:start_node));
167     else
168         % If the index number of the end connection node is larger than
169         % the index of the start connection node, the path will ...
                pass the
170         % max index node and the next node is node with index one.
171         hull_path(path_num).Lat = ...
                fliplr([hull_lat(end_node:num_nodes),...
172             hull_lat(1:start_node)]);
173         hull_path(path_num).Lon = ...
                fliplr([hull_lon(end_node:num_nodes),...
```

```
174                 hull_lon(1:start_node)]);
175         end
176
177         n1 = size(coord(1).Lat,2);
178         n2 = size(coord(2).Lat,2);
179         if n1 > 1
180             lats1 = coord(1).Lat(1:n1-1);
181             lons1 = coord(1).Lon(1:n1-1);
182         else
183             lats1 = coord(1).Lat;
184             lons1 = coord(1).Lon;
185         end
186
187         lats_dest = fliplr(coord(2).Lat);
188         lons_dest = fliplr(coord(2).Lon);
189         if n2 > 1
190             lats2 = lats_dest(2:n2);
191             lons2 = lons_dest(2:n2);
192         else
193             lats2 = lats_dest;
194             lons2 = lons_dest;
195         end
196
197         % The path is is made up from the path from the origin port to the
198         % first connection node, the path along the hull, and the path from
199         % the end connection node to the destination
200         path(path_num).Lat = [lats1, hull_path(path_num).Lat, lats2];
201         path(path_num).Lon = [lons1, hull_path(path_num).Lon, lons2];
202
203         % Generate consistency such that lognitudes are between [-180, 180]
204         % Find nodes with longitudes above 180 degrees, and convert to
205         % negative values
206         above = find(path(path_num).Lon > 180);
207         path(path_num).Lon(above) = path(path_num).Lon(above) - 360;
208
209         below = find(path(path_num).Lon < -180);
210         path(path_num).Lon(below) = path(path_num).Lon(below) + 360;
211
212         % Compute headings and distances for the waypoint legs
213         [¬, distnm] = legs(path(path_num).Lat,path(path_num).Lon,'rh');
214         path(path_num).distnm = distnm;
215         path(path_num).tot_dist = sum(distnm);
216
217         % If polar sailing is not allowed, this rutine multiplies the ...
                length
218         % for such paths
219         if polar == 0
220             artic = find(path(path_num).Lat > 70, 1);
221         else
222             artic = find(path(path_num).Lat > 82, 1);
223         end
224         if isempty(artic) == 0
225             path(path_num).tot_dist = path(path_num).tot_dist*100;
226         end
227     end
228
229     distance = [path(1).tot_dist, path(2).tot_dist];
```

```
230
231 [¬, min_path] = min(distance);
232 [¬, max_path] = max(distance);
233
234 end
```

D.2.13 cross_date_line.m

```
1  function [cross_date, cross_ind] = cross_date_line(lons)
2  % Author:   Ole Brynjar Helland Paulsen
3  % Date:     17.06.2018
4
5  % This function determines if a set of longitude coordinates ...
       describes a
6  % route across the international date line. Returns a variable ...
       indicating
7  % crossing or not, and the index of the node (waypoint) prior to ...
       crossing
8  % if applicable
9
10 % Find if route has positive and negative longitude coordinates
11 a = find(diff(lons>0)≠0);
12 % Sum of absoulute value of coordinates before and after sign change.
13 b = abs(lons(a)) + abs(lons(a+1));
14
15 % If the sum two coordinates are above 180 dregress, the route goes ...
       across
16 % the international date line.
17 if max(b) > 180
18     cross_date = 1;
19     % Indecies of cross date coordinates in sign change index vector
20     c = find(b>180);
21     % Index of node prior to fist crossing of date line
22     d = a(c);
23     cross_ind = d;
24 else
25     cross_date = 0;
26     cross_ind = [];
27 end
28 end
```

D.3 Third level functions

D.3.1 adjacent_nodes.m

```
1  function nodes = adjacent_nodes(node_ID, num_nodes)
2  % Author:   Ole Brynjar Helland Paulsen
3  % Date:     08.06.2018
4
```

```
5  % This function finds the index of the nearest nodes one each side ...
       for a
6  % given node in a closed polygon. If the number of nodes is less ...
       than ten,
7  % only two adjacent nodes are found. Otherwise the four nearest ...
       nodes are
8  % returned.
9
10 if num_nodes < 3
11     error(sprintf('map:%s:inconsistentXY', function_name), ...
           'Function %s requires that the modified hull have at leat ...
           three nodes', upper(function_name));
12 elseif num_nodes < 10
13         if node_ID == 1
14             last = num_nodes;
15             next = node_ID + 1;
16         elseif node_ID == num_nodes
17             last = node_ID - 1;
18             next = 1;
19         else
20             last = node_ID - 1;
21             next = node_ID + 1;
22         end
23         nodes = [last, node_ID, next];
24 else
25     % Number of nodes are greater than 10
26     if node_ID == 1
27         last = num_nodes;
28         last_last = last - 1 ;
29         next = node_ID + 1;
30         next_next = next + 1;
31     elseif node_ID == num_nodes
32         last = node_ID - 1;
33         last_last = last - 1 ;
34         next = 1;
35         next_next = next + 1;
36     elseif node_ID == 2
37         last = node_ID - 1;
38         last_last = num_nodes;
39         next = node_ID + 1;
40         next_next = next + 1;
41     elseif node_ID == num_nodes - 1
42         last = node_ID - 1;
43         last_last = last - 1 ;
44         next = node_ID + 1;
45         next_next = 1;
46     else
47         last = node_ID - 1;
48         last_last = last - 1 ;
49         next = node_ID + 1;
50         next_next = next + 1;
51     end
52
53 nodes = [last_last, last, node_ID, next, next_next];
54 end
```

D.3.2 direction_visi_check.m

```
function [dir] = direction_visi_check(ports, connect_nodes, ...
    num_nodes, hull_lat, hull_lon, inport)
% Author:   Ole Brynjar Helland Paulsen
% Date:     12.03.2020

% Find the direction to travell along the hull path (node index +- ...
    1 ) to
% ensure visibility when original connection node is not visible to the
% corresponding port, by an great circle route.

% Port indices
p = [1,2];

% Define outside port
outport = p(p≠inport);

% Indices of nodes from the two foregoing to the next two.
nodes = adjacent_nodes(connect_nodes(inport), num_nodes);

% Remove current node
midle = round(length(nodes)/2);
nodes(midle) = [];

% Distance from outport towards the nodes
num_n = length(nodes);
dist_n = zeros(num_n,1);
for n = 1:num_n
    lati = hull_lat(nodes(n));
    long = hull_lon(nodes(n));
dist_n(n) = distance('gc',lati, long, ports(outport,1), ...
    ports(outport,2));
end

% Sorted index of distances
[¬, sort_dist] = sort(dist_n);

if sort_dist(1) < midle
    % The shortest path from the outside port is towards the foregoing
    % nodes. Therefore the hull index of the connection node of the ...
        outside
    % port should move in a "-1 direction" if the original node is not
    % visible.
    dir(inport) = 1;
    dir(outport) = -1;
else
    dir(inport) = -1;
    dir(outport) = 1;
end
end
```

D.3.3 nearest_node.m

```
function [node] = nearest_node(hull_lat, hull_lon, y, x)
% Author:   Ole Brynjar Helland Paulsen
% Date:     17.06.2018

% This function returns the index of the polygon node nearest a point
% given by the coordinates [x,y]. The search is limited by only ... searching
% the half, and at most ten, of the hull nodes nearest in longitude to
% the given point.

% Number of different nodes in polygon
num_nodes = length(hull_lat)-1;

% Difference in longitude between hull nodes and the point of
% intersection
diff = abs((hull_lon - x))+ abs((hull_lat-y));

% Original index of the sorted vector
[¬, ind_sort] = sort(diff);

% Search only half or, maximum 10 of the nodes nodes nearest in ... longitude
middle = min([round(num_nodes/2), 30]);
ind_sort = ind_sort(1:middle);

% Seach through half of the nodes to find the nearest in distance
arclen = zeros(middle,1);
for j = 1:middle
    n = ind_sort(j);
    arclen(j) = distance('rh',[y,x],[hull_lat(n), hull_lon(n)]);
end
% Index of shortest distance
[¬, lengt_ind] = min(arclen);
% Hull index
node = ind_sort(lengt_ind);
end
```

D.3.4 nearest_node2.m

```
function [node] = nearest_node(hull_lat, hull_lon, y, x)
% Author:   Ole Brynjar Helland Paulsen
% Date:     17.06.2018

% This function returns the index of the polygon node nearest a point
% given by the coordinates [x,y]. The search is limited by only ... searching
% the half, and at most ten, of the hull nodes nearest in longitude to
% the given point.

% Number of different nodes in polygon
num_nodes = length(hull_lat)-1;

% Difference in longitude between hull nodes and the point of
```

```
% intersection
diff = abs((hull_lon - x))+ abs((hull_lat-y));

% Original index of the sorted vector
[¬, ind_sort] = sort(diff);

% Search only half or, maximum 10 of the nodes nodes nearest in ...
    longitude
middle = min([round(num_nodes/2), 30]);
ind_sort = ind_sort(1:middle);

% Seach through half of the nodes to find the nearest in distance
arclen = zeros(middle,1);
for j = 1:middle
    n = ind_sort(j);
    arclen(j) = distance('rh',[y,x],[hull_lat(n), hull_lon(n)]);
end
% Index of shortest distance
[¬, lengt_ind] = min(arclen);
% Hull index
node = ind_sort(lengt_ind);
end
```

D.3.5 node_index.m

```
function [ind] = node_index(hull_lat, hull_lon, node_lat, node_lon)
% Author:   Ole Brynjar Helland Paulsen
% Date:     08.06.2018

% This function identifies the correct index of a given node in a ...
    polygon.
% The polygon must be closed, such that it starts and ends with the ...
    same
% node.

lat_ind = find(hull_lat == node_lat);
num_lat = length(lat_ind);
lon_ind = find(hull_lon == node_lon);
num_lon = length(lon_ind);
if num_lat > 1 && num_lon > 1
    % node_lat & node _lon occurs several times in polygon. Check ...
        if it is
    % the same entries.
    common = find(lat_ind(:) == lon_ind);
    if length(common) >1
        % node repeats and therefore is the start and end node
        ind = 1;
    else
        % node index from common index of repeating entries
        ind = lat_ind(common);
    end
elseif num_lat > 1
    % More than one node has equal latitude. Therfore the index of the
    % connection node is determined from the longitude coordinates
```

```
    ind = lon_ind;
else
    % Index of node is determined from latitude coordinates
    ind = lat_ind;
end
end
```

D.3.6 split_route.m

```
function [latpts1, lonpts1, latpts2, lonpts2] = split_route(latpts, ...
    lonpts)
% Author:   Ole Brynjar Helland Paulsen
% Date:     08.06.2018

%This function splits a route going across the date line in two parts.
%Returning latitudes and longitudes for the two parts in seperate ...
    vectors.

% Find element before sign change
i =find(diff(lonpts>0)≠0);
n = size(latpts,1);

if size(latpts,2)>1
    latpts = transpose(latpts);
    lonpts = transpose(lonpts);
end

if lonpts(i) > 0    % If track goes from west to east
    nextlon =  (lonpts(i+1) + 360);
    lon1 = 179.9;
    lon2 = 180.1;
else                % If track goes from east to west
    nextlon = (lonpts(i+1) - 360);
    lon1 = -179.9;
    lon2 = -180.1;
end

lat1 = ((latpts(i+1)-latpts(i)) * abs(lon1-lonpts(i)) /...
    abs(nextlon-lonpts(i))) + latpts(i);
lat2 = ((latpts(i+1)-latpts(i)) * abs(lon2-lonpts(i)) /...
    abs(nextlon-lonpts(i))) + latpts(i);

latpts1 = [latpts(1:i); lat1];
lonpts1 = [lonpts(1:i); lon1];
latpts2 = [lat2; latpts(i+1:n)];
lonpts2 = [-lon1; lonpts(i+1:n)];
end
```

Appendix E

MATLAB code for the simulation model

E.1 Overall algorithm

E.1.1 Main script

```
%% Script for running the Simulink simulation model with automatic ...
    route generation

% Author:   Ole Brynjar Helland Paulsen
% Date:     03.05.2020

clear variables
addpath(genpath('..\Simuleringsplatform'))

%% Simulation data. Must be configured by the user.

% Starting date and time. Must be a date within the range covered
% by the given weather data. [yyyy,mm,dd,hh,mm,ss]
start_date = datenum([2016,12,15,06,00,00]);

% Desired time between simluation updates (time between events)
time_step = 1; % [h]

% Sailing speed
speed_sailing = 14.5; %[kts]

%Name of file containing vessel data
file_name_vessel = 'Vessel_Info.xlsx';

%Wind data file
file_wind = 'Weather_data\Wind_northHem_20162017.nc';

```

```
%Wave data file
file_wave = 'Weather_data\Waves_northHem_20162017.nc';

%% Preprocessing
% This section is from the run_master script by Peter Tenfjord and
% Martin Bakke. Read the vessel input file and make the parameters
% available for the Simulink model. All blocks that want to access
% these data: Data Editor-->Input-->Parameter.

% The numbers at the end specify rows in the file, set them according
% to what info you want to use.
xlRange = sprintf('B%d:BC%d',5,5);
[num,txt] = xlsread(file_name_vessel,xlRange);

% Read engine data into EngDat input file. Configure the engine_pre
% function according to your system (Functions folder).
EngDat=engine_pre();

%% Get route

from_file = questdlg('Select option for route data',...
            'Define route input', ...
            'Create new route','Load from file', 'Cancel',...
            'Load from file');

switch from_file
    case 'Load from file'
        uiopen('load')
        run_simulation = 1;
    case 'Create new route'
        % Create new route with the RouteGenerator

        addpath(genpath('..\RouteCreator'))

        % Define origin and destination ports
        [ports, port_path, origin, destination] = select_ports();

        % Get max length of each leg (nm), and option for sailing in
        % polar waters. Polar = 1, when polar sailing is allowed.
        [max_length, polar] = user_input();

        % Generate route
        path = GetRoute(ports, port_path, origin, destination,...
            max_length, polar);

        % Inspect route
        run_simulation = check_route(path);
    case 'Cancel'
        run_simulation = 0;
end

if run_simulation == true
%% Format route parameters for use in the Simulink model

% Waypoint coordinates
wpts = transpose([path.Lat; path.Lon]);

```

```
84  % Number of sailing legs
85  num_legs = size(wpts,1)-1;
86
87  % Courses for each leg
88  course_legs = path.Course;
89
90  % Distances for each leg
91  distnm_legs = path.Distnm;
92
93  % Total distance
94  Total_distance = path.Tot_dist;
95
96  %% Get weather data
97
98  % Add nctoolbox
99  setup_nctoolbox
100
101 % This code must be included to aviod logger warning
102 org.apache.log4j.BasicConfigurator.configure();
103 level = org.apache.log4j.Level.OFF;
104 logger = org.apache.log4j.Logger.getRootLogger();
105 logger.setLevel(level);
106
107 % Maximum time sailing [days], at 65 % of design speed
108 max_sail_time = (Total_distance/(speed_sailing*0.65))/24;
109
110 % End date for weather data subset
111 end_date = start_date + max_sail_time;
112
113 % Load wind and wave data
114 tic
115 [wind_lon, wind_lat, wind_t, wind_u10, wind_v10] ...
116     = wind_data_input(file_wind, path, start_date, end_date);
117
118 [wave_lon, wave_lat, wave_t, wave_mwd, wave_swh, wave_tp] ...
119     = wave_data_input(file_wave, path, start_date, end_date);
120 toc
121
122 %% Run simulation
123 disp('running simulation...')
124 tic
125 sim('Main.slx');
126 toc
127
128 %% Process the output from simulation
129 % This section is partially based on code from the run_master script
130 % by Peter Tenfjord and Martin Bakke. The simulation output is stored
131 % in a structural array, and saved as a matlab file. The time steps,
132 % longitudes and latitudes are updated at the end of each iteration
133 % of the simulation, and have thus one more entry than the other
134 % variables. The other variables have repeating entries at the end,
135 % and this value is therefore not included.
136
137 % Voyage data
138 result.Start = datestr(start_date);                    % Start ...
        time and date
```

```
139 result.End = datestr(start_date + (tout(end)/24));      % End time ...
        and date
140 result.t = tout;                                        % ...
        Simulation clock [h]
141 result.SailTime = tout(end);                            % Time ...
        sailing[h]
142 result.Times = tout(2:end)-tout(1:end-1);               % Time ...
        steps [h]
143 result.LatPts = [LatLon.data(1:end-1,1); wpts(end,1)]; % The ...
        Simulated Latitudes
144 result.LonPts = [LatLon.data(1:end-1,2); wpts(end,2)]; % The ...
        Simulated Longitudes
145 result.Course = Course.data(1:end-1);                   % Course [deg]
146
147 % Met-ocean data
148 result.Hs = MetOcean.data((1:end-1),1);                 % ...
        Significant wave height, Sea based [m]
149 result.Mwd = MetOcean.data((1:end-1),7);                % Mean wave ...
        direction [deg]
150 result.Mwd_rel = MetOcean.data((1:end-1),2);           % Direction ...
        of wave, Sea based  [deg]
151 result.Tp = MetOcean.data((1:end-1),3);                % Peak ...
        period of wave, Sea based [s]
152 result.U10 = MetOcean.data((1:end-1),4);               % Wind ...
        Speed [m/s]
153 result.U10d = MetOcean.data((1:end-1),6);              % Wind ...
        direction [deg]
154 result.U10d_rel = MetOcean.data((1:end-1),5);          % Relative ...
        direction of wind [deg]
155
156 % Attainable speed
157 result.V_att = V_act.data(1:end-1);                    % The ...
        actual attainable speed, from Kwon [kts]
158
159 % Resistance data
160 result.Raw = R.data((1:end-1),1);                      % Added ...
        resistance [N]
161 result.Res = R.data((1:end-1),2);                      % Calm ...
        water resistance [N]
162 result.Raa = R.data((1:end-1),3);                      % Air and ...
        wind resistance [N]
163
164 % Engine data
165 result.ME1FC = engroom.data((1:end-1),1);              % Main ...
        Engine fuel consumption [kg/h]
166 result.ME1PB = engroom.data((1:end-1),2);              % Main ...
        Engine Break Power Production [W]
167 result.ME1X = engroom.data((1:end-1),3);              % Main ...
        Engine Loading degree [%]
168
169
170 % Missing met-ocean data/ NaN nodes
171 NaN_wave = MetOcean.data((1:end-1),8);                 % Number of ...
        nodes used for wave data
172 NaN_wind = MetOcean.data((1:end-1),9);                 % Number of ...
        nodes used for wind data
```

```
173 NaN_wave_ind = find(NaN_wave == 1);                          % Index of ...
        entries with missing wave data
174 NaN_wind_ind = find(NaN_wind == 1);                          % Index of ...
        entries with missing wind data
175
176 %% Process results
177
178 % Correct for missing entries in met-ocean data
179 if isempty(NaN_wave_ind) == 0 || isempty(NaN_wind_ind) == 0
180     result = missing_data(result, NaN_wave, NaN_wave_ind,...
181         NaN_wind, NaN_wind_ind);
182 end
183 clear NaN_wave NaN_wind
184
185 % Derived results
186 result.Max_Hs = max(result.Hs);                             % Max Hs [m]
187 result.Avg_Hs = mean(result.Hs);                            % Average ...
        Hs [m]
188 result.Max_U10 = max(result.U10);                           % Max U10 [m/s]
189 result.Avg_U10 = mean(result.U10);                          % Average ...
        U10 [m/s]
190
191 result.Distance = sum(result.V_att .* result.Times); ...
                          % Distance sailed [nm]
192 result.Avg_V = result.Distance / result.SailTime; ...
                            % Average speed [kts]
193 result.V_set = speed_sailing; ...
                                                  % Desired speed [kts]
194
195 result.Rtot = result.Res + result.Raw + result.Raa; ...
                           % Total resistance [N]
196 result.Raw_Res = (result.Raw./result.Res)*100; ...
                                % Added resistance as a percentage of ...
        calm water resistance [%]
197 result.Avg_Res = sum(result.Rtot.* result.Times) / result.SailTime; ...
          % Average total resistance [N]
198 result.Avg_PB = sum(result.ME1PB.* result.Times) / result.SailTime; ...
          % Average load on Main Engine [W]
199 result.Fuel_Cons_Total = sum(result.ME1FC .* result.Times)/1000; ...
              % Total fuel consumption [t]
200
201 %% Present results and route
202
203 % Create summary table
204 table([
205     cellstr(result.Start);
206     cellstr(result.End);
207     result.SailTime;
208     result.V_set;
209     result.Avg_V;
210     result.Avg_Res/1000;
211     result.Avg_PB/1000;
212     result.Fuel_Cons_Total], 'RowNames', {'Start date', 'End date',...
213     'Time_sailing [h]', 'Set speed [kts]', 'Average speed [kts]', ...
214     'Average_resistance [kN]', 'Average_load_ME [kW]',...
215     'Fuel_consumption [tons]'})
216
```

```
% Plot Simulation results
plot_results;

% Plot route
plot_route(path,result);

% Save result
answer = questdlg('Save results?',...
            'Save', ...
            'Save','Cancel', 'Save');
if strcmp(answer,'Save')
    uisave('result','simulation_output')
end
snapnow;

else
% Display warning in the case of non-feasible route
f = msgbox({'The process will terminate without running a ...
    simulation', ...
    'Please try first creating a route, and then load route from ...
        file.'}...
    , 'Warning');
waitfor(f)
close all
end
```

E.1.2 MATLAB functions

The script for running simulations includes a function for establishing engine data kept from the original model, which is not included here.

check_route.m

```
function run_simulation = check_route(path)
% Author:   Ole Brynjar Helland Paulsen
% Date:     03.12.2018

% This function displays a route on a map, and asks the user to verify
% the fitness of the route for running a simulation. A variable
% indication whether or not to proceed with the simulation is returned.

% Intermediate resolution shorelines
load map_data.mat map_data

% Latitude coordinates
lats = path.Lat;
% Longitude coordinates
lons = path.Lon;

ports = [lats(1), lons(1);
    lats(end), lons(end)];

%% Setting latitude and longitude limits for the maps
```

```
%Minimum and maximum latitude of paths
min_lat = min(lats);
max_lat = max(lats);

if path.Cross_date == 0
    % The route is not crossing the date line
    min_lon = min(lons);
    max_lon = max(lons);

    % Longitudinal span
    lon_span = max_lon - min_lon;

else
    ind = path.Cross_ind;

    % The route goes across the international date line
    if length(ind) == 1 && lons(ind) > 0 % route goes from west to east
        lon_west = lons(1:ind);
        lon_east = lons((ind+1):end);
    elseif length(ind) == 1
        %Route goes from east to west
        lon_west = lons((ind+1):end);
        lon_east = lons(1:ind);
    elseif lons(ind(1)) > 0 % Route crossing from west to east ...
        initially
        lon_west = [lons(1:ind(1)), lons((ind(2)+1):end)];
        lon_east = lons((ind(1)+1):ind(2));
    else
        %Route crossing from east to west initially
        lon_west = lons((ind(1)+1):ind(2));
        lon_east = [lons(1:ind(1)), lons((ind(2)+1):end)];
    end
    min_lon = min(lon_west);
    max_lon = max(lon_east);

    %Longitudinal span
    lon_span = (max_lon+360) - min_lon;
end

% Add buffer to map limits in case of route going in opposite direction
% of the destination ports for a section of the route
lon_buffer = 40;
mlon_min = min_lon - lon_buffer/2;
mlon_max = max_lon + lon_buffer/2;

% Set latitude limits for mercantor map

% Minimum latitude span for visual purposes
lat_span_min = (9/16) * (lon_span+lon_buffer);
lat_diff = max_lat - min_lat;
if lat_diff < lat_span_min
    extend = max([((lat_span_min - lat_diff)*0.5), 4]);
else
    extend = 4;
end
mlat_min = max([min_lat - extend, -85]);
mlat_max = min([max_lat + extend, 85]);
```

```
77
78
79  % Label spacing for mercantor map
80  if lon_span > 300
81      label_space = 45;
82  elseif lon_span > 100
83      label_space = 30;
84  else
85      label_space =15;
86  end
87
88  %% Generate Marcantor map
89
90  figure('color','w');
91  ha = axesm('mapproj','mercator',...
92             'maplatlim',[mlat_min mlat_max],'maplonlim',...
93             [mlon_min mlon_max]);
94  axis off, gridm on, framem on;
95  setm(ha,'MLineLocation',15,'PLineLocation',15);
96  mlabel on, plabel on;
97  setm(gca,'MLabelLocation',label_space)
98  geoshow([map_data.Lat], [map_data.Lon])
99  geoshow( 'worldcities.shp' , 'Marker' , '.' , 'Color' , 'red' )
100
101 %Display ports
102 geoshow(ports(1,1),ports(1,2),'DisplayType','point',...
103 'markeredgecolor','k','markerfacecolor','k','marker','o')
104 textm(ports(1,1),ports(1,2), '  Origin')
105 geoshow(ports(2,1),ports(2,2),'DisplayType','point',...
106 'markeredgecolor','k','markerfacecolor','k','marker','o')
107 textm(ports(2,1),ports(2,2), '  Destination')
108
109 % Display paths
110 geoshow(lats, lons,'DisplayType','line',...
111                          'color','green','linestyle','-')
112 % Verify route
113 f = msgbox('Please inspeckt the route, then press OK to proceed',...
114     'Verify route');
115 waitfor(f)
116  answer = questdlg('Proceed simulation with route as shown?', ...
117     'Verify route', ...
118     'Proceed','Cancel','Proceed');
119 if strcmp(answer, 'Proceed') == 1
120     run_simulation = 1;
121 else
122     run_simulation = 0;
123 end
124 end
```

wind_data_input.m

```
1  function [wind_lon, wind_lat, wind_t, wind_u10, wind_v10] ...
2      = wind_data_input(file_wind, path, start_date, end_date)
3  % Author:   Ole Brynjar Helland Paulsen
```

```
4  % Date:     17.06.2018
5
6  % This function reads wind data from a specified NetCDF file (.nc),
7  % containing the five vaiables loaded below. The variable names ...
       used in
8  % the given file must correspond to the variables used here, i.e. 'u10'
9  % and 'v10'. The longitudes of the wind data must be ordered in
10 % acending order.
11 disp('loading wind data from file...')
12
13 % Read wind data from nc file
14 nc = ncgeodataset(file_wind);
15
16 %% Time data
17 % Read time variable. Time in the current weather data is given in
18 % hours since Jan. 1. 1900, 00:00.
19 t = nc.geovariable('time');
20 wind_t = double(t.data(:));
21 clear t
22
23 % Resolution of time data
24 time_res = wind_t(2) - wind_t(1);
25
26 % MATLAB functions such as datenum() denotes time as days since
27 % Jan. 1. 0000, 00:00. Therefore the weather data must be manipulated
28 % to correspond to the MATLAB format
29 weather_start = datenum([1900,1,1,00,00,00]);
30 wind_t = (wind_t / 24) + weather_start;
31
32 % Time range covered by weather data
33 min_time_data = wind_t(1);
34 max_time_data = wind_t(end);
35
36 % Check if data covers the desired time period for the simulation
37 if start_date < min_time_data || end_date > max_time_data
38     disp('The given weather data does not cover the given time ...
           period of the simulation')
39 end
40
41 % Lower bound time range from the date closest to the start date
42 T1 = abs(wind_t - start_date);
43 [¬, t1] = min(T1);
44 clear T1
45
46 % Ensure start date is after lowest time bound
47 if start_date < wind_t(t1)
48     t1 = t1 - 1;
49 end
50
51 % Upper bound time range
52 T2 = abs(wind_t - end_date);
53 [¬, t2] = min(T2);
54 clear T2
55
56 if end_date > wind_t(t2)
57     t2 = t2 + 1;
58 end
```

```
59
60  % If the wind data has time reolution of 1 hour, the data is collected
61  % for every other time step in order to reduce the size of the
62  if time_res ≤ 1
63      steps = 2;
64      if mod(t2-t1,2) == 0
65          % Number of entries covering the time range is even numbered.
66          % Must add an extra element to cover the end date.
67          t2=t2+1;
68      end
69  else
70      steps = 1;
71  end
72
73  wind_t = wind_t(t1:steps:t2);
74
75  %% Latitudes
76  % Read latitude data
77  wind_lat = nc.data('latitude');
78  wind_lat = double(wind_lat);
79
80  % Size of latidude grid
81  lat_grid_size = abs(wind_lat(2)-wind_lat(1));
82
83  % Check if wind data covers the latitude range of the route:
84
85  % Range of latitudes included in the route
86  min_lat_route = min(path.Lat);
87  max_lat_route = max(path.Lat);
88
89  % Latitudes covered by wind data
90  min_lat_data = min(wind_lat);
91  max_lat_data = max(wind_lat);
92
93  if min_lat_route < min_lat_data || max_lat_route > max_lat_data
94      disp('The given weather data does not cover the given route')
95      fprintf('The route covers latitudes from %d to %d, while the ...
            wind data covers latitudes from %d to %d'...
96          , min_lat_route, max_lat_route, min_lat_data, max_lat_data)
97  end
98
99  % Find latitudes to include:
100
101 % Latitude range
102 % Index of nearest wind latitude for min latitude
103 L1 = abs(wind_lat - min_lat_route);
104 [¬, lat_min] = min(L1);
105 clear L1
106
107 % Index of nearest wind latitude for max latitude
108 L2 = abs(wind_lat - max_lat_route);
109 [¬, lat_max] = min(L2);
110 clear L2
111
112 if wind_lat(1) > wind_lat(2)
113     % Weather latitudes ordered in decending order.
114     lat1 = lat_max;
```

```
115      lat2 = lat_min;
116      % Ensure extreme values of route latitude is included:
117      if max_lat_route > wind_lat(lat1)
118          lat1 = lat1 - 1;
119      end
120      if min_lat_route < wind_lat(lat2)
121          lat2 = lat2 + 1;
122      end
123  else
124      % Weather latitudes ordered in acending order.
125      lat1 = lat_min;
126      lat2 = lat_max;
127      % Ensure extreme values of route latitude is included:
128      if min_lat_route < wind_lat(lat1)
129          lat1 = lat1 - 1;
130      end
131      if max_lat_route > wind_lat(lat2)
132          lat2 = lat2 + 1;
133      end
134  end
135
136  % Check resolution of latitude grid and number of entries included
137  if lat_grid_size < 0.4 && (lat2-lat1) > 400
138      lat_step = 2; % Extract every other element
139      if mod((lat2-lat1),2) == 0  % Size is even number
140          lat2 = lat2 + 1;        % Include an extra element
141      end
142  else
143      lat_step = 1;
144  end
145
146  % Extract relevant entries
147  wind_lat = wind_lat(lat1:lat_step:lat2);
148
149  %% Longitudes
150  % Read longitude data
151  wind_lon = nc.data('longitude');
152  wind_lon = double(wind_lon);
153
154  % Size of longitude grid
155  lon_grid_size = wind_lon(2)-wind_lon(1);
156
157  % Longitudes covered by wind data
158  min_lon_data = min(wind_lon);
159  max_lon_data = max(wind_lon);
160
161  if path.Cross_date == 0
162      % Route does not cross the date line
163
164      % Check if wind data covers the longitude range of the route
165      min_lon_route = min(path.Lon);
166      max_lon_route = max(path.Lon);
167      if min_lon_route < min_lon_data || max_lon_route > max_lon_data
168      disp('The given weather data does not cover the given route')
169      fprintf('The route covers longitudes from %d to %d, while the ...
             wind data covers longitudes from %d to %d'...
170          , min_lon_route, max_lon_route, min_lon_data, max_lon_data)
```

```
171     end
172
173     % Find relevant longitudes to include:
174
175     % Index of nearest wind lonitude for min longitude
176     Long1 = abs(wind_lon - min_lon_route);
177     [¬, lon1] = min(Long1);
178     clear Long1
179     % Ensure extreme values of route latitude is included
180     if min_lon_route < wind_lon(lon1)
181         lon1 = lon1 - 1;
182     end
183
184     % Index of nearest wind lonitude for max longitude
185     Long2 = abs(wind_lon - max_lon_route);
186     [¬, lon2] = min(Long2);
187     clear Long2
188     % Ensure extreme values of route latitude is included
189     if max_lon_route > wind_lon(lon2)
190         lon2 = lon2 + 1;
191     end
192
193     % Check for longitude grid size and number of entries.
194     if lon_grid_size < 0.4 && (lon2-lon1) > 400
195         lon_step = 2; % Extract only every other entry
196          if mod((lon2-lon1),2) == 0     % size is even number
197             lon2=lon2+1;        % include an extra element
198          end
199     else
200         lon_step = 1;
201     end
202
203     % Extract relevant longitudes
204     wind_lon = wind_lon(lon1:lon_step:lon2);
205 else
206     % The route goes across the international date line
207
208     % Range of longitudes included in the route
209     lon_cell_size = abs(wind_lon(2) - wind_lon(1));
210     buffer  = lon_cell_size + 0.3;
211     if (buffer - 180) < min_lon_data || (180 - buffer) > max_lon_data
212     disp('Please check if the given wind data covers the longitude ...
            range og the desired route')
213     fprintf('The wind data covers longitudes from %d to %d',...
214         min_lon_data, max_lon_data)
215     end
216
217     % Find relevant longitudes to include. Find logitudes east and west
218     % of the date line seperatly, before joing them in one vector:
219
220     % Longitude coordinates of route
221     lons = path.Lon;
222     % Index of node(s) before route goes across the date line
223     ind = path.Cross_ind;
224
225     % Seperate route longitudes in eastern and western part. East and
226     % west referes to direction of route, not east and west on a map.
```

```
227     if length(ind) == 1 && lons(ind) > 0
228         % Route goes from west to east
229         lon_west = lons(1:ind);
230         lon_east = lons((ind+1):end);
231     elseif length(ind) == 1
232         % Route goes from east to west
233         lon_west = lons((ind+1):end);
234         lon_east = lons(1:ind);
235     elseif lons(ind(1)) > 0
236         % Route crossing from west to east initially
237         lon_west = [lons(1:ind(1)), lons((ind(2)+1):end)];
238         lon_east = lons((ind(1)+1):ind(2));
239     else
240         % Route crossing from east to west initially
241         lon_west = lons((ind(1)+1):ind(2));
242         lon_east = [lons(1:ind(1)), lons((ind(2)+1):end)];
243     end
244 
245     % Longitude of eastern point of route
246     east_lon_route = max(lon_east);
247 
248     % Index of nearest wind lonitude for the eastward point of the ...
            route
249     Long1 = abs(wind_lon - east_lon_route);
250     [¬, lon1] = min(Long1);
251     clear Long1
252 
253     % Check if nearest wind longitude is east of eastern point of route
254     if east_lon_route > wind_lon(lon1)
255         lon1 = lon1 + 1;
256     end
257 
258     % Longitude of western point of route
259     west_lon_route = min(lon_west);
260 
261     % Index of nearest wind lonitude for the westward point of the ...
            route
262     Long2 = abs(wind_lon - west_lon_route);
263     [¬, lon2] = min(Long2);
264     clear Long2
265 
266     % Check if nearest wind longitude is west of western point of route
267     if west_lon_route < wind_lon(lon2)
268         lon2 = lon2 - 1;
269     end
270 
271     % Check for longitude grid size and number of entries.
272     if lon_grid_size < 0.4 && (lon1 + length(wind_lon(lon2:end)) > 400)
273         lon_step = 2; % Extract only every other entry
274         if mod(lon1,2) == 0
275             % Number of entries east of date line is even numbered
276             lon1=lon1+1;        % Include an extra element
277         end
278         if mod(length(wind_lon(lon2:end)),2) == 0
279             % Number of entries west of date line is even numbered
280             lon2=lon2-1;        % Include an extra element
281         end
```

```
282     else
283         lon_step = 1;
284     end
285     % Extract relevant longitudes
286     wind_lons_east = wind_lon(1:lon_step:lon1);
287     wind_lons_west = wind_lon(lon2:lon_step:end);
288 
289     % Combine longitues in one variable
290     wind_lon = [wind_lons_east; wind_lons_west];
291     clear wind_lons_east wind_lons_west
292 end
293 
294 %% Wind data
295 % Dimesions are (time,lat,lon)
296 
297 if path.Cross_date == 0
298     % 10 metre U wind component
299     wind_u10 = nc{'u10'}(t1:steps:t2, lat1:lat_step:lat2,...
300         lon1:lon_step:lon2);
301 
302     % 10 metre V wind component
303     wind_v10 = nc{'v10'}(t1:steps:t2, lat1:lat_step:lat2,...
304         lon1:lon_step:lon2);
305 else
306     num_lons = nc.size('longitude');
307 
308     % 10 metre U wind component
309     u1 = nc{'u10'}(t1:steps:t2, lat1:lat_step:lat2,...
310         1:lon_step:lon1);
311     u2 = nc{'u10'}(t1:steps:t2, lat1:lat_step:lat2,...
312         lon2:lon_step:num_lons);
313     wind_u10 = cat(3, u1, u2);
314     clear u1 u2
315 
316     % 10 metre V wind component
317     v1 = nc{'v10'}(t1:steps:t2, lat1:lat_step:lat2,...
318         1:lon_step:lon1);
319     v2 = nc{'v10'}(t1:steps:t2, lat1:lat_step:lat2,...
320         lon2:lon_step:num_lons);
321     wind_v10 = cat(3, v1, v2);
322 end
323 disp('Wind data loaded')
324 end
```

wave_data_input.m

```
1  function [wave_lon, wave_lat, wave_t, wave_mwd, wave_swh, wave_tp] ...
2      = wave_data_input(file_wave, path, start_date, end_date)
3  % Author:   Ole Brynjar Helland Paulsen
4  % Date:     17.06.2018
5  
6  % This function reads wave data from a specified NetCDF file (.nc),
7  % containing the six vaiables loaded below. The variable names used in
8  % the given file must correspond to the variables used here, i.e.
```

```
% 'mwd', 'swh' and 'pp1d'. The longitudes of the wave data must be
% ordered in ascending order.

disp('Loading wave data from file...')
% Read wave data from nc files
wave = ncgeodataset(file_wave);

%% Time data
% Read time variable. Time in the current weather data is given in
% hours since Jan. 1. 1900, 00:00.
t = wave.geovariable('time');
wave_t = double(t.data(:));
clear t

% Resolution of time data
time_res = wave_t(2) - wave_t(1);

% MATLAB functions such as datenum() denotes time as days since
% Jan. 1. 0000, 00:00. Therefore the weather data must be manipulated
% to correspond to the MATLAB format
weather_start = datenum([1900,1,1,00,00,00]);
wave_t = (wave_t / 24) + weather_start;

% Time range covered by weather data
min_time_data = wave_t(1);
max_time_data = wave_t(end);

% Check if data covers the desired time period for the simulation
if start_date < min_time_data || end_date > max_time_data
    disp('The given weather data does not cover the given time ...
        period of the simulation')
end

% Lower bound time range from the date closest to the start date
T1 = abs(wave_t - start_date);
[¬, t1] = min(T1);
clear T1

% Ensure start date is after lowest time bound
if start_date < wave_t(t1)
    t1 = t1 - 1;
end

% Upper bound time range
T2 = abs(wave_t - end_date);
[¬, t2] = min(T2);
clear T2

if end_date > wave_t(t2)
    t2 = t2 + 1;
end

% If the wind data has time reolution of 1 hour, the data is collected
% for every other time step in order to reduce the size of the
% variables
if time_res ≤ 1
    steps = 2;
```

```
65      if mod(t2-t1,2) == 0
66          % Number of entries covering the time range is even numbered.
67          % Must add an extra element to cover the end date.
68          t2=t2+1;
69      end
70  else
71      steps = 1;
72  end
73
74  wave_t = wave_t(t1:steps:t2);
75
76  %% Latitudes
77  % Read latitude data
78  wave_lat = wave.data('latitude');
79  wave_lat = double(wave_lat);
80
81  % Size of latidude grid
82  lat_grid_size = abs(wave_lat(2)-wave_lat(1));
83
84  %Check if wave data covers the latitude range of the route:
85
86  % Range of latitudes included in the route
87  min_lat_route = min(path.Lat);
88  max_lat_route = max(path.Lat);
89
90  % Latitudes covered by wave data
91  min_lat_data = min(wave_lat);
92  max_lat_data = max(wave_lat);
93
94  if min_lat_route < min_lat_data || max_lat_route > max_lat_data
95      disp('The given weather data does not cover the given route')
96      fprintf('The route covers latitudes from %d to %d, while the ...
            wave data covers latitudes from %d to %d', min_lat_route, ...
            max_lat_route, min_lat_data, max_lat_data)
97  end
98
99  % Find latitudes to include:
100
101 % Latitude range
102 % Index of nearest wave latitude for min latitude
103 L1 = abs(wave_lat - min_lat_route);
104 [¬, lat_min] = min(L1);
105 clear L1
106
107 % Index of nearest wave latitude for max latitude
108 L2 = abs(wave_lat - max_lat_route);
109 [¬, lat_max] = min(L2);
110 clear L2
111
112 if wave_lat(1) > wave_lat(2)
113     % Weather latitudes ordered in decending order.
114     lat1 = lat_max;
115     lat2 = lat_min;
116     % Ensure extreme values of route latitude is included:
117     if max_lat_route > wave_lat(lat1)
118         lat1 = lat1 - 1;
119     end
```

```
120      if min_lat_route < wave_lat(lat2)
121          lat2 = lat2 + 1;
122      end
123  else
124      % Weather latitudes ordered in acending order.
125      lat1 = lat_min;
126      lat2 = lat_max;
127      % Ensure extreme values of route latitude is included:
128      if min_lat_route < wave_lat(lat1)
129          lat1 = lat1 - 1;
130      end
131      if max_lat_route > wave_lat(lat2)
132          lat2 = lat2 + 1;
133      end
134  end
135
136  % Check resolution of latitude grid and number of entries included
137  if lat_grid_size < 0.4 && (lat2-lat1) > 400
138      lat_step = 2; % Extract every other element
139      if mod((lat2-lat1),2) == 0  % Size is even number
140          lat2 = lat2 + 1;        % Include an extra element
141      end
142  else
143      lat_step = 1;
144  end
145
146  wave_lat = wave_lat(lat1:lat_step:lat2);
147
148  %% Longitudes
149  % Read longitude data
150  wave_lon = wave.data('longitude');
151  wave_lon = double(wave_lon);
152
153  % Size of longitude grid
154  lon_grid_size = wave_lon(2)-wave_lon(1);
155
156  % Longitudes covered by wave data
157  min_lon_data = min(wave_lon);
158  max_lon_data = max(wave_lon);
159
160  if path.Cross_date == 0
161      % Route does not cross the date line
162
163      % Check if wave data covers the longitude range of the route
164      min_lon_route = min(path.Lon);
165      max_lon_route = max(path.Lon);
166      if min_lon_route < min_lon_data || max_lon_route > max_lon_data
167      disp('The given weather data does not cover the given route')
168      fprintf('The route covers longitudes from %d to %d, while the ...
             wave data covers longitudes from %d to %d', min_lon_route, ...
             max_lon_route, min_lon_data, max_lon_data)
169      end
170
171      % Find relevant longitudes to include:
172
173      %Index of nearest wave lonitude for min longitude
174      Long1 = abs(wave_lon - min_lon_route);
```

```
175       [¬, lon1] = min(Long1);
176       clear Long1
177       %Ensure extreme values of route latitude is included
178       if min_lon_route < wave_lon(lon1)
179           lon1 = lon1 - 1;
180       end
181
182       %Index of nearest wave lonitude for max longitude
183       Long2 = abs(wave_lon - max_lon_route);
184       [¬, lon2] = min(Long2);
185       clear Long2
186       %Ensure extreme values of route latitude is included
187       if max_lon_route > wave_lon(lon2)
188           lon2 = lon2 + 1;
189       end
190
191       % Check for longitude grid size and number of entries.
192       if lon_grid_size < 0.4 && (lon2-lon1) > 400
193           lon_step = 2; % Extract only every other entry
194            if mod((lon2-lon1),2) == 0     % size is even number
195               lon2=lon2+1;        % include an extra element
196            end
197       else
198           lon_step = 1;
199       end
200
201       % Extract relevant longitudes
202       wave_lon = wave_lon(lon1:lon_step:lon2);
203   else
204       % The route goes across the international date line
205
206       % Range of longitudes included in the route
207       lon_cell_size = abs(wave_lon(2) - wave_lon(1));
208       buffer  = lon_cell_size + 0.3;
209       if (buffer - 180) < min_lon_data || (180 - buffer) > max_lon_data
210       disp('Please check if the given wave data covers the longitude ...
              range og the desired route')
211       fprintf('The wave data covers longitudes from %d to %d',...
212           min_lon_data, max_lon_data)
213       end
214
215       % Find relevant longitudes to include. Find logitudes east and west
216       % of the date line seperatly, before joing them in one vector:
217
218       % Longitude coordinates of route
219       lons = path.Lon;
220       % Index of node(s) before route goes across the date line
221       ind = path.Cross_ind;
222
223       % Seperate route longitudes in eastern and western part. East and
224       % west referes to direction of route, not east and west on a map.
225       if length(ind) == 1 && lons(ind) > 0
226           % Route goes from west to east
227           lon_west = lons(1:ind);
228           lon_east = lons((ind+1):end);
229       elseif length(ind) == 1
230           % Route goes from east to west
```

```
231         lon_west = lons((ind+1):end);
232         lon_east = lons(1:ind);
233     elseif lons(ind(1)) > 0
234         % Route crossing from west to east initially
235         lon_west = [lons(1:ind(1)), lons((ind(2)+1):end)];
236         lon_east = lons((ind(1)+1):ind(2));
237     else
238         % Route crossing from east to west initially
239         lon_west = lons((ind(1)+1):ind(2));
240         lon_east = [lons(1:ind(1)), lons((ind(2)+1):end)];
241     end
242 
243     % Longitude of eastern point of route
244     east_lon_route = max(lon_east);
245 
246     % Index of nearest wave lonitude for the eastward point of the ...
            route
247     Long1 = abs(wave_lon - east_lon_route);
248     [¬, lon1] = min(Long1);
249     clear Long1
250 
251     % Check if nearest wave longitude is east of eastern point of route
252     if east_lon_route > wave_lon(lon1)
253         lon1 = lon1 + 1;
254     end
255 
256     % Longitude of western point of route
257     west_lon_route = min(lon_west);
258 
259     % Index of nearest wave lonitude for the westward point of the ...
            route
260     Long2 = abs(wave_lon - west_lon_route);
261     [¬, lon2] = min(Long2);
262     clear Long2
263 
264     % Check if nearest wave longitude is west of western point of route
265     if west_lon_route < wave_lon(lon2)
266         lon2 = lon2 - 1;
267     end
268 
269     % Check for longitude grid size and number of entries.
270     if lon_grid_size < 0.4 && (lon1 + length(wave_lon(lon2:end)) > 400)
271         lon_step = 2; % Extract only every other entry
272         if mod(lon1,2) == 0
273             % Number of entries east of date line is even numbered
274             lon1=lon1+1;        % Include an extra element
275         end
276         if mod(length(wave_lon(lon2:end)),2) == 0
277             % Number of entries west of date line is even numbered
278             lon2=lon2-1;        % Include an extra element
279         end
280     else
281         lon_step = 1;
282     end
283     % Extract relevant longitudes
284     wave_lons_east = wave_lon(1:lon_step:lon1);
285     wave_lons_west = wave_lon(lon2:lon_step:end);
```

```
286
287     % Combine longitues in one variable
288     wave_lon = [wave_lons_east; wave_lons_west];
289     clear wind_lons_east wind_lons_west
290 end
291
292 %% Wave data
293 % Dimesions are (time,lat,lon)
294
295 if path.Cross_date == 0
296     % Mean wave direction
297     wave_mwd = wave{'mwd'}(t1:steps:t2, lat1:lat_step:lat2,...
298         lon1:lon_step:lon2);
299
300     % Significant wave height
301     wave_swh = wave{'swh'}(t1:steps:t2, lat1:lat_step:lat2,...
302         lon1:lon_step:lon2);
303
304     % Period
305     wave_tp = wave{'pp1d'}(t1:steps:t2, lat1:lat_step:lat2,...
306         lon1:lon_step:lon2);
307 else
308     num_lons = wave.size('longitude');
309
310     % Mean wave direction
311     mwd1 = wave{'mwd'}(t1:steps:t2, lat1:lat_step:lat2,...
312         1:lon_step:lon1);
313     mwd2 = wave{'mwd'}(t1:steps:t2, lat1:lat_step:lat2,...
314         lon2:lon_step:num_lons);
315     wave_mwd = cat(3, mwd1, mwd2);
316     clear mwd1 mwd2
317
318     % Significant wave height
319     swh1 = wave{'swh'}(t1:steps:t2, lat1:lat_step:lat2,...
320         1:lon_step:lon1);
321     swh2 = wave{'swh'}(t1:steps:t2, lat1:lat_step:lat2,...
322         lon2:lon_step:num_lons);
323     wave_swh = cat(3, swh1, swh2);
324     clear swh1 swh2
325
326     % Period
327     tp1 = wave{'pp1d'}(t1:steps:t2, lat1:lat_step:lat2,...
328         1:lon_step:lon1);
329     tp2 = wave{'pp1d'}(t1:steps:t2, lat1:lat_step:lat2,...
330         lon2:lon_step:num_lons);
331     wave_tp = cat(3, tp1, tp2);
332 end
333 disp('wave data loaded')
334 end
```

missing_data.m

```
1 function result = missing_data(result, NaN_wave, NaN_wave_ind, ...
      NaN_wind, NaN_wind_ind)
```

```
% Author:   Ole Brynjar Helland Paulsen
% Date:     01.03.2019

% This function handles missing entries in the met-ocean data. If data
% is missing for an event, the data is replaced with the value of the
% previous event. If data is missing for the first event, the entry is
% replaced with the value of the first event containing data.

if isempty(NaN_wave_ind) == 0 && isempty(NaN_wind_ind) == 0
    % Both wave and wind data have missing entries
    % Index of simulation steps where either wind data, wave data ...
        or both
    % are missing.
    NaN_ind = unique(sort([NaN_wave_ind;NaN_wind_ind]));
    NotaN = NaN_wave;
    NotaN(NaN_ind)=0;
elseif isempty(NaN_wave_ind) == 0
    % Wave data has missing entries
    NotaN = NaN_wave;
else
    % Wind data has missing entries
    NotaN = NaN_wind;
end

%% Adjust parameters that are dependent on both wave and wind data

% First entry containing data
first = find(NotaN ≠ 0,1);

if first > 1
    % First enty is missing data. Replace entry with value of next
    % enrty containg data
    result.ME1FC(1) = result.ME1FC(first);
    result.ME1PB(1) = result.ME1PB(first);
    result.ME1X(1) = result.ME1X(first);
    NotaN(1) = 4;
end
% Index of remaining entries missing data
NaN_ind = find(NotaN==0);

while isempty(NaN_ind) == 0
    % Replace entries with value of previous entry
    result.ME1FC(NaN_ind) = result.ME1FC(NaN_ind-1);
    result.ME1PB(NaN_ind) = result.ME1PB(NaN_ind-1);
    result.ME1X(NaN_ind) = result.ME1X(NaN_ind-1);
    NotaN(NaN_ind) = NotaN(NaN_ind-1);

    NaN_ind = find(NotaN==0);
end

clear NaN
%% Wave data and dependent parameters
if isempty(NaN_wave_ind) == 0
    disp('Wave data has missing entries')
    if NaN_wave_ind(1) == 1
    % The first entry is missing

```

```
58      first_wave = find(NaN_wave ≠ 0,1);
59      % Replace first entry of wave data and wave dependent data with
60      % the value of the first entry containing data
61      result.Hs(1) = result.Hs(first_wave);
62      result.Mwd(1) = result.Mwd(first_wave);
63      result.Mwd_rel(1) = result.Mwd_rel(first_wave);
64      result.Tp(1) = result.Tp(first_wave);
65      result.Raw(1) = result.Raw(first_wave);
66      result.Res(1) = result.Res(first_wave);
67      NaN_wave(1) = 4;
68      end
69
70      NaN_wave_ind = find(NaN_wave == 0);
71
72      while isempty(NaN_wave_ind) == 0
73          % Replace missing entries with value of previous entry
74          result.Hs(NaN_wave_ind) = result.Hs(NaN_wave_ind-1);
75          result.Mwd(NaN_wave_ind) = result.Mwd(NaN_wave_ind-1);
76          result.Mwd_rel(NaN_wave_ind) = result.Mwd_rel(NaN_wave_ind-1);
77          result.Tp(NaN_wave_ind) = result.Tp(NaN_wave_ind-1);
78          result.Raw(NaN_wave_ind) = result.Raw(NaN_wave_ind-1);
79          result.Res(NaN_wave_ind) = result.Res(NaN_wave_ind-1);
80          NaN_wave(NaN_wave_ind) = NaN_wave(NaN_wave_ind-1);
81
82          NaN_wave_ind = find(NaN_wave == 0);
83      end
84  end
85      %% Wind data
86  if isempty(NaN_wind_ind) == 0
87      disp('Wind data has missing entries')
88      if NaN_wind_ind(1) == 1
89      % The first entry is missing
90
91      first_wind = find(NaN_wind ≠ 0,1);
92      % Replace first entry of wind data and wind dependent data with
93      % the value of the first entry containing data
94      result.U10(1) = result.U10(first_wind);
95      result.U10d(1) = result.U10d(first_wind);
96      result.U10d_rel(1) = result.U10d_rel(first_wind);
97      result.Raa(1) = result.Raa(first_wind);
98      NaN_wind(1) = 4;
99      end
100
101     NaN_wind_ind = find(NaN_wind == 0);
102
103     while isempty(NaN_wind_ind) == 0
104         % Replace missing entries with value of previous entry
105         result.Hs(NaN_wind_ind) = result.Hs(NaN_wind_ind-1);
106         result.Mwd(NaN_wind_ind) = result.Mwd(NaN_wind_ind-1);
107         result.Mwd_rel(NaN_wind_ind) = result.Mwd_rel(NaN_wind_ind-1);
108         result.Tp(NaN_wind_ind) = result.Tp(NaN_wind_ind-1);
109         result.Raw(NaN_wind_ind) = result.Raw(NaN_wind_ind-1);
110         result.Res(NaN_wind_ind) = result.Res(NaN_wind_ind-1);
111         NaN_wind(NaN_wind_ind) = NaN_wind(NaN_wind_ind-1);
112
113         NaN_wind_ind = find(NaN_wind == 0);
114     end
```

```
115 end
116 end
```

plot_results.m

```
1  % Author:   Ole Brynjar Helland Paulsen
2  % Date:     03.12.2018
3
4  % Script for plotting some simulation results
5
6  % Time steps
7  t = result.t(1:end-1);
8
9  %% Plot break power from main engine
10
11 % Break power for main engine in kW
12 PB = result.ME1PB / 1000;
13
14 x = length(t);
15 y_PB = max(PB) + 500;
16
17 figure
18 plot(t,(PB),'DisplayName','PBME1')
19 title('Break Power from Main Engine')
20 xlim([0 x]);
21 ylim([0 y_PB]);
22 legend('hide')
23 xlabel('Time Step [h]')
24 ylabel('Break Power [kW]')
25
26 clear PB
27
28 %% Plot attainable speed, wave height and direction.
29
30 % Attainable speeds
31 V_att = result.V_att;
32
33 % Hs
34 Hs = result.Hs;
35
36 % Relative wave direction
37 mwd = abs(wrapTo180(result.Mwd_rel));
38
39 figure
40 subplot(3,1,1)
41 plot(t,V_att,'DisplayName','V_att')
42 title('Attainable speed during voyage')
43 xlim([0 x]);
44 legend('hide')
45 xlabel('Time Step [h]')
46 ylabel('Attainable Speed [kts]')
47 dim = [0.7 0.63 0.1 0.2];
48 str = sprintf('Set Speed = %.2f',speed_sailing);
49 annotation('textbox',dim,'String',str,'FitBoxToText','on')
```

```
50
51  subplot(3,1,2)
52  plot(t,Hs,'DisplayName', 'Hs')
53  title('Encountered Hs during voyage')
54  xlim([0 x]);
55  legend('hide')
56  xlabel('Time Step [h]')
57  ylabel('Significant Wave Height [m]')
58
59  subplot(3,1,3)
60  plot(t,mwd,'DisplayName','Mwd')
61  title('Mean wave direction relative to course during voyage (0-180)')
62  xlim([0 x]);
63  legend('hide')
64  xlabel('Time Step [h]')
65  ylabel('Relative Direction [deg]')
66
67  clear mwd
68
69  %%  Plot total resistance and added resistance
70
71  % Resistance
72  Rtot = result.Rtot;
73  Res = result.Res;
74  Raw = result.Raw;
75  Raa = result.Raa;
76
77  figure
78  plot(t,Rtot/1000,'DisplayName','RTot')
79  hold on
80  plot(t,Res/1000,'DisplayName','Rcw')
81  plot(t,Raw/1000,'DisplayName','Raw')
82  plot(t,Raa/1000,'DisplayName','Raa')
83  title('Total Resistance and Resistance Components')
84  xlim([0 x]);
85  xlabel('Time Step [h]')
86  ylabel('Resistance [kN]')
87  legend('show')
88  dim = [0.7 0.35 0.1 0.2];
89  str = sprintf('Set Speed = %.2f \n Speed Loss: On',speed_sailing);
90  annotation('textbox',dim,'String',str,'FitBoxToText','on')
91  hold off
92
93  clear Rtot Res Raw
94
95  %% Plot wind data and resistance
96
97  % Wind direction
98  U10d_rel = result.U10d_rel;
99
100 %Wind speed in direction of ship
101 Ux = result.U10.*(cosd(U10d_rel));
102
103 % Wind speed relative to ship (positive = head wind)
104 U = Ux + V_att*1852/3600;
105
106 figure
```

```
107 subplot(3,1,1)
108 plot(t,Raa/1000,'DisplayName','Raa')
109
110 title('Resistance due to air and wind')
111 xlim([0 x]);
112 xlabel('Time Step [h]')
113 ylabel('Resistance [kN]')
114 legend('hide')
115 hold off
116
117 subplot(3,1,2)
118 plot(t,V_att*1852/3600,'DisplayName', 'Attainable speed')
119 hold on
120 plot(t,Ux,'DisplayName','Wind speed component in direction of ship')
121 plot(t,U,'DisplayName','Wind speed relative to ship')
122 title('Attainable ship speed and wind velocity in direction of ship')
123 xlim([0 x]);
124 legend('show')
125 xlabel('Time Step [h]')
126 ylabel('Velocity [m/s]')
127 hold off
128
129 subplot(3,1,3)
130 plot(t, abs(wrapTo180(U10d_rel)), 'DisplayName',...
131     'Relative wind direction')
132 title('Wind direction relative to course (0 = head wind)')
133 xlim([0 x]);
134 legend('hide')
135 xlabel('Time Step [h]')
136 ylabel('Direction off ship course [deg]')
137
138 clear Raa V_att U10d_rel Ux U
139
140 %% Plot wave data
141
142 % Plot Hs
143 figure
144 h = histogram(Hs,'DisplayName','Hs');
145 title('Distribution of encountered wave heights')
146 legend('hide')
147 xlabel('Hs [m]')
148 ylabel('Percent')
149 h.Normalization= 'probability';
150 h.BinWidth=1;
151 ytix = get(gca, 'YTick');
152 set(gca, 'YTick',ytix, 'YTickLabel',ytix*100)
153 clear Hs
154
155 % Wave period
156 Tp = result.Tp;
157
158 % Plot histogram of wave periods encountered
159 figure
160 h2=histogram(Tp,'DisplayName','Tp Sea');
161 title('Distribution of encountered wave periods')
162 legend('hide')
163 xlabel('Tp [s]')
```

```
ylabel('Percent')
ylim([0 0.2])
h2.Normalization= 'probability';
h2.BinWidth=1;
ytix = get(gca, 'YTick');
set(gca, 'YTick',ytix, 'YTickLabel',ytix*100)
clear Tp
```

plot_route.m

```
function plot_route(path,result)
% Author:   Ole Brynjar Helland Paulsen
% Date:     10.06.2020

% This function draws an orthogonal projection map of the route
% described by the "path" variable. Also an animation of the simulated
% voyage is created, from the result variable given as input.

addpath('..\RouteCreator\Input')

% Intermediate resolution shorelines
load map_data.mat map_data

% Latitude coordinates
lats = path.Lat;
% Longitude coordinates
lons = path.Lon;

% Port coordinates
ports = [lats(1), lons(1);
    lats(end), lons(end)];

% Latidues and longitudes for simulated voyage
LatPts = result.LatPts;
LonPts = result.LonPts;

%% Setting latitude and longitude limits for the maps
%Minimum and maximum latitude of paths
min_lat = min(lats);
max_lat = max(lats);

%Calculate latitude midpoint
lat_origo = min_lat + (max_lat - min_lat)/2;

if path.Cross_date == 0
    % The route is not crossing the date line
    min_lon = min(lons);
    max_lon = max(lons);

    % Longitudinal span
    lon_span = max_lon - min_lon;

    % Origo of orthogonal map
    lon_origo = min_lon + lon_span/2;
```

```
45
46  else
47      % The route goes across the international date line
48      % Index of node(s) prior to crossing
49      ind = path.Cross_ind;
50
51      if length(ind) == 1 && lons(ind) > 0 % route goes from west to east
52          lon_west = lons(1:ind);
53          lon_east = lons((ind+1):end);
54      elseif length(ind) == 1
55          %Route goes from east to west
56          lon_west = lons((ind+1):end);
57          lon_east = lons(1:ind);
58      elseif lons(ind(1)) > 0 % Route crossing from west to east ...
            initially
59          lon_west = [lons(1:ind(1)), lons((ind(2)+1):end)];
60          lon_east = lons((ind(1)+1):ind(2));
61      else
62          %Route crossing from east to west initially
63          lon_west = lons((ind(1)+1):ind(2));
64          lon_east = [lons(1:ind(1)), lons((ind(2)+1):end)];
65      end
66      min_lon = min(lon_west);
67      max_lon = max(lon_east);
68
69      %Longitudinal span
70      lon_span = (max_lon+360) - min_lon;
71
72      % Origo of orthogonal map
73      if abs(max_lon)>abs(min_lon)
74          lon_origo = min_lon + lon_span/2;
75      else
76          lon_origo = max_lon - lon_span/2;
77      end
78  end
79
80  % Add buffer to map limits
81  lon_buffer = 24;
82  mlon_min = min_lon - lon_buffer/2;
83  mlon_max = max_lon + lon_buffer/2;
84
85  % Set latitude limits for mercantor map
86
87  % Minimum latitude span for visual purposes
88  lat_span_min = (9/16) * (lon_span+lon_buffer);
89  lat_diff = max_lat - min_lat;
90  if lat_diff < lat_span_min
91      extend = max([((lat_span_min - lat_diff)*0.5), 4]);
92  else
93      extend = 4;
94  end
95  mlat_min = max([min_lat - extend, -85]);
96  mlat_max = min([max_lat + extend, 85]);
97
98
99  % Label spacing for mercantor map
100 if lon_span > 300
```

```
    label_space = 45;
elseif lon_span > 100
    label_space = 30;
else
    label_space =15;
end

%% Generate ortholgonal map
figure('color','w');
ha_o = axesm('mapproj', 'ortho', 'origin',[lat_origo lon_origo]);
axis off, gridm on, framem on;
setm(ha_o,'MLineLocation',15,'PLineLocation',15);
mlabel on, plabel on;
mlabel('equator')
plabel(lon_origo-45);
plabel('fontweight','bold')
setm(gca,'MLabelLocation',30, 'PLabelLocation',15)
geoshow([map_data.Lat], [map_data.Lon],'color','black')
geoshow('landareas.shp', 'FaceColor', [0.15 0.5 0.15])
geoshow( 'worldcities.shp' , 'Marker' , '.' , 'Color' , 'red')

%Display ports
geoshow(ports(1,1),ports(1,2),'DisplayType','point',...
'markeredgecolor','k','markerfacecolor','k','marker','o')
textm(ports(1,1),ports(1,2), '  Origin')
geoshow(ports(2,1),ports(2,2),'DisplayType','point',...
'markeredgecolor','k','markerfacecolor','k','marker','o')
textm(ports(2,1),ports(2,2), '  Destination')

% Compute points for great circle route.
gcpts = track2('gc',ports(1,1),ports(1,2),ports(2,1), ports(2,2));

% Display great circle route
geoshow(gcpts(:,1),gcpts(:,2),'DisplayType','line',...
    'color','cyan','linestyle','--')

% Display path
geoshow(lats, lons,'DisplayType','line',...
                        'color','b','linestyle','-')

%% Generate Mercantor map and show animation

figure('color','w');
ha = axesm('mapproj','mercator','maplatlim',...
    [mlat_min mlat_max],'maplonlim',[mlon_min mlon_max]);
axis off, gridm on, framem on;
setm(ha,'MLineLocation',15,'PLineLocation',15);
mlabel on, plabel on;
setm(gca,'MLabelLocation',label_space)
geoshow([map_data.Lat], [map_data.Lon],'color','black')
geoshow('landareas.shp', 'FaceColor', [0.15 0.5 0.15])
geoshow( 'worldcities.shp' , 'Marker' , '.' , 'Color' , 'red' )

%Display ports
geoshow(ports(1,1),ports(1,2),'DisplayType','point',...
'markeredgecolor','k','markerfacecolor','k','marker','o')
textm(ports(1,1),ports(1,2), '  Origin')
```

```
158 geoshow(ports(2,1),ports(2,2),'DisplayType','point',...
159 'markeredgecolor','k','markerfacecolor','k','marker','o')
160 textm(ports(2,1),ports(2,2), '  Destination')
161
162 % Display great circle route
163 geoshow(gcpts(:,1),gcpts(:,2),'DisplayType','line',...
164     'color','cyan','linestyle','--')
165
166 %Show animation of Voyage
167 [x,y] = mfwdtran(LatPts,LonPts);
168 comet(x,y);
169
170 end
```

E.2 Simulink functions

The simulation model includes five additional Simulink functions that were kept from the original model. These are not included here.

Initial value of simulation variables

```
1  function [sim_date, lat, lon] = initialize(start_date, wpts)
2  % Author:   Ole Brynjar Helland Paulsen
3  % Date:     03.12.2018
4
5  % Setting the simulation date as the given start date
6  sim_date = start_date;
7
8  % Setting latitude and longitude from the first waypoint
9  lat = wpts(1,1);
10 lon = wpts(1,2);
11 end
```

Course and distance of new leg

```
1  function [course, dist_leg] = new_leg(leg, course_legs, distnm_legs)
2  % Author:   Ole Brynjar Helland Paulsen
3  % Date:     03.12.2018
4
5  course = course_legs(leg);
6  dist_leg = distnm_legs(leg);
7  end
```

Met-ocean data for current event

```
function [tp, hs, rel_mwd, wind_speed, rel_wind_dir, wind_dir, mwd, ...
    missing_wave_data, missing_wind_data] = ocean_data(lat, lon, ...
    sim_date, course, wave_lon, wave_lat, wave_t, wind_lon, ...
    wind_lat, wind_t, wave_swh, wave_mwd, wave_tp, wind_u10, wind_v10)
% Author:   Ole Brynjar Helland Paulsen
% Date:     03.12.2018

% Function for setting values of met-ocaen data variables from
% provided data set. Values are interpolated from the up to four
% nearest locations and two closest time entries.
coder.extrinsic('deg2nm')
coder.extrinsic('distance')
coder.extrinsic('intersect')
coder.extrinsic('find')
coder.extrinsic('length')
coder.extrinsic('wrapTo180')

% Pre allocate variables
num_nodes_waves = [0,0];
num_nodes_wind = [0,0];
num_lons_wave = 0;
num_lons_wind = 0;
missing_wave_data = 0;
missing_wind_data = 0;

tp_t = [0,0];
hs_t = [0,0];
mwd_t = [0,0];
u10_t = [0,0];
v10_t = [0,0];
rel_mwd = 0;
rel_wind_dir = 0;

%% Time coordinates
% Wave data time coordinate closest to the current time:
[¬,wave_t1] = min(abs(wave_t - sim_date));

% Second closest time coordinate:
if sim_date < wave_t(wave_t1)
    wave_t2 = wave_t1 - 1;
else
    wave_t2 = wave_t1 + 1;
end

wave_t_ind = [wave_t1, wave_t2];

% Wind data time coordinate closest to the current time:
[¬, wind_t1] = min(abs(wind_t - sim_date));

% Next closest time coordinate:
if sim_date < wind_t(wind_t1)
    wind_t2 = wind_t1 - 1;
else
    wind_t2 = wind_t1 + 1;
end

wind_t_ind = [wind_t1, wind_t2];
```

```
55
56  %% Longitudes
57  % Identify the two wave data longitude coordinates nearest the
58  % longitude of the current waypoint
59
60  % Index of nearest longitude
61  [¬, ind_lon_wave1] = min(abs(wave_lon-lon));
62
63  % Index of the next to nearest longitude. The longitudes in the weather
64  % data spans from [-180, 179.7], but the code accounts for the ...
        possibility
65  % that the first longitude is larger than -180 deg.
66
67  lon_nearest_wave = wave_lon(ind_lon_wave1);
68  num_lons_wave = length(wave_lon);
69
70  if ind_lon_wave1 == 1 && lon <  lon_nearest_wave
71      ind_lon_wave2 = num_lons_wave;
72  elseif ind_lon_wave1 == num_lons_wave && lon > lon_nearest_wave
73      ind_lon_wave2 = 1;
74  elseif lon  < lon_nearest_wave
75      ind_lon_wave2 = ind_lon_wave1 - 1;
76  else
77      ind_lon_wave2 = ind_lon_wave1 + 1;
78  end
79
80  % Index of nearest wind longitude
81  [¬, ind_lon_wind1] = min(abs(wind_lon-lon));
82
83  % Index of the next to nearest wind longitude
84  lon_nearest_wind = wind_lon(ind_lon_wind1);
85  num_lons_wind = length(wind_lon);
86
87  if ind_lon_wind1 == 1 && lon <  lon_nearest_wind
88      ind_lon_wind2 = num_lons_wind;
89  elseif ind_lon_wind1 == num_lons_wind && lon > lon_nearest_wind
90      ind_lon_wind2 = 1;
91  elseif lon  < lon_nearest_wind
92      ind_lon_wind2 = ind_lon_wind1 - 1;
93  else
94      ind_lon_wind2 = ind_lon_wind1 + 1;
95  end
96
97  %% Latitudes
98
99  %Index of nearest wave latitude
100 [¬, ind_lat_wave1] = min(abs(wave_lat - lat));
101
102 %Index of next to nearest wave latitude
103 lat_nearest_wave = wave_lat(ind_lat_wave1);
104 if lat  < lat_nearest_wave
105     ind_lat_wave2 = ind_lat_wave1 + 1;
106 else
107     ind_lat_wave2 = ind_lat_wave1 - 1;
108 end
109
110 %Index of nearest wind latitude
```

```
111 [¬, ind_lat_wind1] = min(abs(wind_lat - lat));
112
113 %Index of next to nearest wind latitude
114 lat_nearest_wind = wind_lat(ind_lat_wind1);
115 if lat  < lat_nearest_wind
116     ind_lat_wind2 = ind_lat_wind1 + 1;
117 else
118     ind_lat_wind2 = ind_lat_wind1 - 1;
119 end
120
121 %% Wave data
122
123 % Coordinates of the 4 nearest nodes of wave data
124 nodes_wave = [wave_lat(ind_lat_wave1), wave_lon(ind_lon_wave1);
125     wave_lat(ind_lat_wave1), wave_lon(ind_lon_wave2);
126     wave_lat(ind_lat_wave2), wave_lon(ind_lon_wave1);
127     wave_lat(ind_lat_wave2), wave_lon(ind_lon_wave2)];
128 % Indices of the wave data nodes stored in matrix
129     node_ind_wave = [ind_lat_wave1, ind_lon_wave1;
130     ind_lat_wave1, ind_lon_wave2;
131     ind_lat_wave2, ind_lon_wave1;
132     ind_lat_wave2, ind_lon_wave2];
133
134 for t = 1:2
135     % Check if the data is NaN at any of the 4 nodes, for each ...
          points in time.
136     % It is assumed that if data is missing for one variable, all ...
          variables are
137     % missing. Therfore only one variable is checked for each data set.
138     TP = [1000;1000;1000;1000];
139     for p=1:4
140         % Value of Tp data at each coordinate
141         tp_i = wave_tp(wave_t_ind(t), node_ind_wave(p,1), ...
              node_ind_wave(p,2));
142         % Store the data value in the varable vector if data value ...
              is a
143         % number
144         if isnan(tp_i) == 0
145             TP(p) = tp_i;
146         end
147     end
148
149     %Indices of non-NaN enties for each time step. NaN entries will ...
          have the
150     %inital defined value of 1000.
151     non_NaN_TP = find(TP≠1000);
152
153     % Number of nodes kept
154     num_nodes_waves(t) = length(non_NaN_TP);
155
156     % Indices of the non-NaN entries/nodes
157     num_NaN_TP = zeros(num_nodes_waves(t),1);
158     num_NaN_TP = non_NaN_TP;
159
160     % New variables for storing wave nodes.
161     if num_nodes_waves(t) < 4
```

```
162         % It is neccessary to pre allocate variables in order to ...
                not get a
163         % dimension mismatch error.
164         wave_nodes = zeros(num_nodes_waves(t),2);
165         wave_nodes_ind = zeros(num_nodes_waves(t),2);
166         wave_nodes = nodes_wave(num_NaN_TP,:);
167         wave_nodes_ind = node_ind_wave(num_NaN_TP,:);
168     else
169         wave_nodes = nodes_wave;
170         wave_nodes_ind = node_ind_wave;
171     end
172
173     % Distance from the current waypoint to each of the adjacent nodes
174     arclen_waves = zeros(num_nodes_waves(t),1);
175     for i = 1:num_nodes_waves(t)
176     dist = distance('gc',lat, lon,wave_nodes(i,1), wave_nodes(i,2));
177     arclen_waves(i) = deg2nm(dist);
178     end
179
180     % Normalised weighting of each of the nearest nodes based on ...
            nearest
181     % distace
182     weight_coord = (sum(arclen_waves) ./arclen_waves) / ...
            sum((sum(arclen_waves) ./arclen_waves) );
183
184     % Interpolated values of wave period, mean wave direction and ...
            significant
185     % wave height
186     for p=1:num_nodes_waves(t)
187             tp_t(t) = tp_t(t)+ wave_tp(wave_t_ind(t), ...
                    wave_nodes_ind(p,1), wave_nodes_ind(p,2))* ...
                    weight_coord(p);
188             hs_t(t) = hs_t(t) + wave_swh(wave_t_ind(t), ...
                    wave_nodes_ind(p,1), wave_nodes_ind(p,2))* ...
                    weight_coord(p);
189             mwd_t(t) = mwd_t(t) +  wave_mwd(wave_t_ind(t), ...
                    wave_nodes_ind(p,1), wave_nodes_ind(p,2))* ...
                    weight_coord(p);
190     end
191 end
192
193 if num_nodes_waves(1) > 0 && num_nodes_waves(2) > 0 %isnan(tp_t(1)) ...
        == 0 && isnan(tp_t(2)) == 0
194     % Both point in time have available data from at least one node.
195
196     % Relative weighting of the two time indices:
197     t_dist_wave = abs(sim_date - [wave_t(wave_t1), wave_t(wave_t2)]);
198     weight_time_wave = 1 - (t_dist_wave / sum(t_dist_wave));
199
200     % Parameters interpolated between two closest time points
201     tp = tp_t(1)* weight_time_wave(1) + tp_t(2)* weight_time_wave(2);
202     hs = hs_t(1)* weight_time_wave(1) + hs_t(2)* weight_time_wave(2);
203     mwd = mwd_t(1)* weight_time_wave(1) + mwd_t(2)* ...
            weight_time_wave(2);
204 elseif num_nodes_waves(1) == 0 && num_nodes_waves(2) == 0
205     % Data is missing for all node and both points in time. Assigning
206     % arbitrary values to wave parameters. Will be replaced in
```

```
207      % post-processing
208      tp = 10;
209      hs = 3;
210      mwd = 90;
211      missing_wave_data = 1;
212      disp('wave data missing')
213  elseif num_nodes_waves(1) == 0
214      % Data is missing for all nodes for nearest point in time. Data
215      % available for other time point is used.
216      tp = tp_t(2);
217      hs = hs_t(2);
218      mwd = mwd_t(2);
219  else
220      % Data missing for second closest time point. Values for ...
             closest point
221      % in time is used.
222      tp = tp_t(1);
223      hs = hs_t(1);
224      mwd = mwd_t(1);
225  end
226
227  % Relative direction of incoming waves, 0 degrees = head waves.
228  adjust = 180; % Direction convention: where waves are travelling ...
         towards.
229  % adjust = 0; % Direction convention: where waves are coming from.
230  rel_mwd = abs(wrapTo180(mwd - course + adjust));
231
232  %% Wind data
233
234  % Coordinates of the 4 nearest nodes for the wind data
235      nodes_wind = [wind_lat(ind_lat_wind1), wind_lon(ind_lon_wind1);
236      wind_lat(ind_lat_wind1), wind_lon(ind_lon_wind2);
237      wind_lat(ind_lat_wind2), wind_lon(ind_lon_wind1);
238      wind_lat(ind_lat_wind2), wind_lon(ind_lon_wind2)];
239  % Indices of the wind data nodes stored in matrix
240      node_ind_wind = [ind_lat_wind1, ind_lon_wind1;
241      ind_lat_wind1, ind_lon_wind2;
242      ind_lat_wind2, ind_lon_wind1;
243      ind_lat_wind2, ind_lon_wind2];
244
245  for t=1:2
246      U10 = [1000;1000;1000;1000];
247      % Check if the data is NaN at any of the 4 nodes and 2 points ...
             in time.
248      for p=1:4
249          % Value of u10 data at each coordinate
250          u10_i = wind_u10(wind_t_ind(t), node_ind_wind(p,1), ...
                 node_ind_wind(p,2));
251          % Store the data value in the varable vector if data value ...
                 is a
252          % number
253          if isnan(u10_i) == 0
254              U10(p) = u10_i;
255          end
256      end
257
```

```
258     %Indices of non-NaN enties for each time step. NaN entries will ...
            have the
259     %inital defined value of 1000.
260     non_NaN_U10 = find(U10≠1000);
261
262     % Number of nodes kept
263     num_nodes_wind(t) = length(non_NaN_U10);
264
265     % Indices of the non-NaN entries/nodes
266     num_NaN_U10 = zeros(num_nodes_wind(t),1);
267
268     num_NaN_U10 = non_NaN_U10;
269
270     % New variables for storing wind nodes.
271     if num_nodes_wind(t) < 4
272         wind_nodes = zeros(num_nodes_wind(t),2);
273         wind_nodes_ind = zeros(num_nodes_wind(t),2);
274         wind_nodes = nodes_wind(num_NaN_U10,:);
275         wind_nodes_ind = node_ind_wind(num_NaN_U10,:);
276         disp('less than four nodes used for wind data')
277         disp(num_nodes_wind(t))
278     else
279         wind_nodes = nodes_wind;
280         wind_nodes_ind = node_ind_wind;
281     end
282
283     % Distance from the current waypoint to each of the adjacent nodes
284     arclen_wind = zeros(num_nodes_wind(t),1);
285     for j = 1:num_nodes_wind(t)
286     dist = distance('gc',lat, lon,wind_nodes(j,1), wind_nodes(j,2));
287     arclen_wind(j) = deg2nm(dist);
288     end
289
290     % Normalised weighting of each of the nearest nodes based on ...
            nearest
291     % distace
292     weight_coord_wind = (sum(arclen_wind) ./arclen_wind) / ...
            sum((sum(arclen_wind) ./arclen_wind) );
293
294     % Interpolated values of 10 metres u and v wind components.
295     for p=1:num_nodes_wind(t)
296         u10_t(t) = u10_t(t) + wind_u10(wind_t_ind(t), ...
                wind_nodes_ind(p,1), wind_nodes_ind(p,2))* ...
                weight_coord_wind(p);
297         v10_t(t) = v10_t(t) + wind_v10(wind_t_ind(t), ...
                wind_nodes_ind(p,1), wind_nodes_ind(p,2))* ...
                weight_coord_wind(p);
298     end
299
300 end
301
302 if num_nodes_wind(1) > 0 && num_nodes_wind(2) > 0 %isnan(u10_t(1)) ...
        == 0 && isnan(u10_t(2)) == 0
303     % Both point in time have available data from at least one node.
304     % Relative weighting of the two time indices:
305     t_dist_wind = abs(sim_date - [wind_t(wind_t1), wind_t(wind_t2)]);
306     weight_time_wind = 1 - (t_dist_wind / sum(t_dist_wind));
```

```
307
308     % Wind speeds interpolated between the two cosest points in time
309     u10 = u10_t(1) * weight_time_wind(1) + u10_t(2) * ...
            weight_time_wind(2);
310     v10 = v10_t(1) * weight_time_wind(1) + v10_t(2) * ...
            weight_time_wind(2);
311 elseif num_nodes_wind(1) == 0 && num_nodes_wind(2) == 0
312     % Data is missing for all node and both points in time. Assigning
313     % arbitrary values to wave parameters. Will be replaced in
314     % post-processing
315     u10 = 10;
316     v10 = 10;
317     missing_wind_data = 1;
318     disp('wind data missing')
319 elseif num_nodes_wind(1) == 0
320     % Data is missing for all nodes for nearest point in time. Data
321     % available for other time point is used.
322     disp('wind data missing for one time point')
323     u10 = u10_t(2);
324     v10 = v10_t(2);
325 else
326     % Data missing for second closest time point. Values for ...
            closest point
327     % in time is used.
328     disp('wind data missing for one time point')
329     u10 = u10_t(1);
330     v10 = v10_t(1);
331 end
332 % Resultant wind speed from components
333 wind_speed = sqrt(u10^2 + v10^2);
334
335 % Wind direction in degrees from u-direction
336 wd = atand(v10/u10);
337
338 % Direction from north meridian [0-360 degrees]
339 if u10 > 0
340     wind_dir = 90 - wd;
341 else
342     wind_dir = 270 - wd;
343 end
344
345 % Wind direction i relative to the ship course. Degrees (0-360) of ...
        CL, head
346 % wind = 0, tail wind = 180.
347 rel_wind_dir = wrapTo180(wind_dir - course);
348 rel_wind_dir = rel_wind_dir + 180;
349 end
```

Combined air and wind resistance

```
1 function R_AA = air_res(Vreq,u10,u10d)
2 % Author:   Ole Brynjar Helland Paulsen
3 % Date:     03.12.2018
4
```

```
% Function for calculating combined wind and air resistance

rho_air = 1.25; %kg/m3 density, air
Ap = 700; % m2 transverse projected area above water
C_air = 0.60; % Drag coefficient

% Speed of air equal to ship speed [m/s]
V = Vreq*1852/3600;

% Wind speed in direction of ship. Posivite speed for head wind, ...
    negative
%for tail wind.
Ux = u10*(cosd(u10d));

U = Ux + V; % Wind speed relative to vessel

% Resistance due to air and wind
R_AA = C_air*0.5*rho_air*(U*abs(U))*Ap;

% Incoming wind speed assumed to be uniform

end
```

Planning next event

```
function [lat_new, lon_new, time_next_event, distance_next, ...
    complete_leg] = update_sim(speed,leg,lat,lon, dist_sailed_leg, ...
    course, dist_leg, wpts, time_step)
% Author:   Ole Brynjar Helland Paulsen
% Date:     03.12.2018

% Function responsibel for planning the next event, by finding the
% corrdinates of the next event, as well as the distance to and time
% until the next event.
coder.extrinsic('nm2deg')
coder.extrinsic('reckon')

% It is neccesary to give initial value to the lat and lon vaiable
% to aviod error in assigned dimensions in Simulink.
lat_new = 0;
lon_new = 0;

% Distance sailed [nm]
distance_next = speed * time_step;

% Remaining distance of current leg
remain_dist_leg = dist_leg - dist_sailed_leg;

if distance_next < remain_dist_leg
    % Next event is a point on the same leg

    % Distance sailed in given timestep[deg]
    arc_length=nm2deg(distance_next);

```

```
28      % Coordinates of point reached in a given timestep along the
29      % course of the current leg
30      [lat_new, lon_new]=reckon('rh',lat, lon, arc_length, course);
31
32      %Time until next event
33      sailing_time = time_step;
34
35      % Leg is not complete
36      complete_leg = 0;
37  else
38      % Next event starts a new leg
39
40      % Coordinates
41      lat_new = wpts((leg+1),1);
42      lon_new = wpts((leg+1),2);
43
44      % Time until next waypoint at current speed
45      sailing_time = (remain_dist_leg / speed);
46
47      % Distace to next event/waypoint
48      distance_next = remain_dist_leg;
49
50      % Next event marks the completion of the current leg
51      complete_leg = 1;
52  end
53  % Time until next event
54  time_next_event = sailing_time;
55  end
```

Check if journey is completed

```
1   function outport = fcn(Leg, num_legs)
2   % Author:   Ole Brynjar Helland Paulsen
3   % Date:     03.12.2018
4
5   if Leg == num_legs
6       % Journey is complete
7       outport = 1;
8   else
9       % Jouney is not complete
10      outport=2;
11  end
12
13  end
```

Ole Brynjar H. Paulsen

Routing module in a discrete-event simulation platform

NTNU
Norwegian University of
Science and Technology