

TMR 4345 Marin Datalab

Project Report

Ship Rounting

Weijian Yang

546843

May 2021

Department of Marine Technology



NTNU – Trondheim
Norwegian University of
Science and Technology

Contents

1	Introduction	1
1.1	Background	1
1.2	Objective	2
1.3	Structure of the report	2
2	Methodology	3
2.1	Cost calculating	3
2.2	Resistance	4
2.2.1	Conventions	4
2.2.2	ITTC-Wind	5
2.2.3	Hollenbach	6
2.2.4	STAwave-1	9
2.3	Path planning	9
2.3.1	Dijkstra	10
3	Programming process	12
3.1	Python	12
3.1.1	Packages	12
3.2	Assumptions	13
3.3	Programming process	13
3.3.1	Data processing	14
3.3.2	Cost calculation	14
3.3.3	Path finding	16
3.3.4	Velocity picking	18
4	Modelling	19
4.1	Parameters for testing	19
4.2	Tests and results	20
5	Conclusion	24
5.1	Conclusion	24
5.2	Reflection and future work	24
A	Codes	26
A.1	Functions	26
A.2	Testing codes	43

Chapter 1

Introduction

1.1 Background

It is little or no alternative for the vast majority of trade over the world to transport by ship, in which case ship routing is a significant task during the voyage. With this task, ship could follow a safe and efficient path which is helpful for improving delivery efficiency and shortening delivery time. So, efficient ship routing is meaningful for both shipping itself and the entire industry chain. However, in recent years, it is very hard to keep improving the performance of ship engines. Instead, finding an efficient path would play a more important role in reducing fuel consumption. Moreover, the concept of the ‘green shipping’ has been raised in the international shipping industry and within the foreseeable future, shipping will still be dependent on fossil fuels. Therefore, an optimum path means not only economic efficiency but also environment protection to prevent pollution.

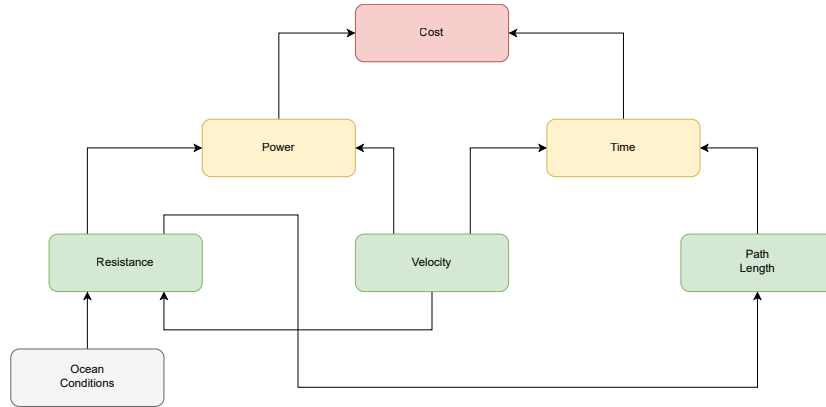


Figure 1.1: Cost analysis

For a single voyage, the trip cost and its causes need to be discussed. Usually, this cost would be defined as the fuel consumption or the needed energy in this voyage, and it would be affected by two parameters, power per moment and task time. To calculate power, we need know the velocity and resistance of

each moment. On the other side, the ship resistance would be affected by the velocity as well. Apart from ship velocity, the ocean environment is the other reason we need to take into account in resistance calculation, because these conditions would change if the ship follow different paths. Other elements, like ship parameters, are also important for resistance, but in this case, we could do nothing on these elements. The time to complete the task depends on the velocity of ship and the length of chosen path. To choose this path, the resistance is almost the most significant basis. In conclusion, as shown in Figure 1.1, environment conditions, ship velocities, resistance and path length must be discussed to find an efficient path.

1.2 Objective

In this project, a ship needs to transport load from point A to point B within a given time, in which case the optimal route, with respect to fuel consumption given some weather data, need to be shown. For this aim, costs of different places under different velocities and different heading angles would be calculated by various methods, like Hollenbach method, STAwave-1 method and ITTC method. Then, based on these data, total costs of different path would be calculated and compared by Dijkstra method. Then for each velocity, there would be one most efficient path, which could be compared again to find the optimal velocity causing minimum cost. To sum up, after several comparisons, the optimal route, the efficient velocity and the minimum cost would be found.

1.3 Structure of the report

Chapter 2 would tell more details about different methods used in this project. Chapter 3 would introduce the programming tool and process, which would illustrate some important assumptions and the function of different codes.

Chapter 4 would contain some tests and discussion.

Chapter 5 would provide conclusion and recommendations for further work.

Chapter 2

Methodology

In this chapter, the cost function would be defined at first. Then, owing to its importance, different methods of calculating resistance would be introduced. In final, the theory about path planning would be announced. However, in this chapter, only theories would be shown. In programming process, these methods or equations would be modified based on some assumptions, which would be elaborated in chapter 3.

2.1 Cost calculating

In this project, we could estimate the cost function as the energy needed for a path. So, the cost function could be stated as:

$$C = \int_0^t P(t)dt \quad (2.1)$$

- C : Cost of voyage
- P : Power of ship engine
- t : Time needed in this task

As we known, there will be resistance when the ship moves forward. To keep the desired velocity, we need thrust to counteract this resistance and the effective power of ship engine could be given as:

$$P_E = T_e * V_G = R * V_G \quad (2.2)$$

- P_E : Effective power of ship engine
- T_e : Effective thrust
- V_G : Measured ship's speed over ground
- R : Resistance of shipping

Actually, this effective power P_E is different from the power of engine P , whose relationship could be rewritten as:

$$P_E = \eta P \quad (2.3)$$

- η : Conversion efficiency

This conversion efficiency would be affected by many factors like ship velocities, engine and shafting structure and work environment.

The resistance in Eq.(2.2) is related to many parameters as well, like ship velocities, ship parameters and ocean conditions. It would be elaborated in section 2.2. On the other hand, the task time Eq.(2.1) is related with the path length and the ship velocity, so it could be written as:

$$t = L_{path}/V_G \quad (2.4)$$

- L_{path} : Length of chosen path

In conclusion, the problem of this project could be modelled as that, an appropriate path whose length is L_i , and efficient ship velocities $V_{G,j}$ in each part of this path would be found to make the cost function of this voyage be minimum.

$$\min_{i,j} C_{i,j} = f[P(R_{i,j}, V_{G,j}), t(L_{path,i}, V_{G,j})]$$

2.2 Resistance

Ship is a kind of vehicle travelling between water and air, which means that the resistance of shipping could be classified as two parts based on different causes. The resistance R_{wind} which caused by wind could be calculated by the ITTC recommended method. In some places, this resistance R_{wind} could be considered as a part of added resistance, but in this project, it is split as an independent part because of its unique calculation method. The interaction between hull and calm water leads to bare-hull resistance R_{hull} , which consists many kinds of resistance, like viscous resistance, friction resistance and wave resistance. In this project, Hollenbach method was used to calculate R_{hull} . In addition, STAwave-1 was utilized to take added resistance caused by head waves R_{wave} into this project.

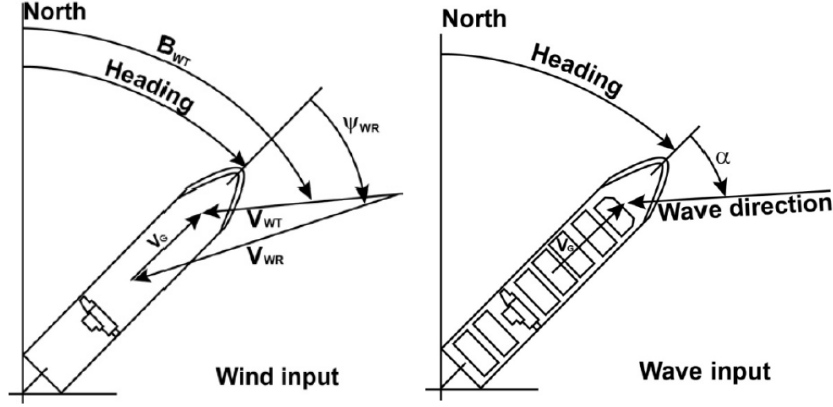
So, in this project, the total resistance R_{total} could be classified by these 3 independent parts:

$$R_{total} = R_{wind} + R_{hull} + R_{wave} \quad (2.5)$$

- R_{total} : Total resistance
- R_{wind} : Added resistance caused by wind
- R_{hull} : Bare-hull resistance in calm water, including viscous resistance, friction resistance and wave resistance
- R_{wave} : Added resistance caused by waves in advance

2.2.1 Conventions

It is important to define sign conventions, which could illustrate directions of ship, wave and wind. According to ITTC [1], these conventions are shown as:



(a) Sign convention for wind directions (b) Sign convention for wave directions

Figure 2.1: Sign conventions

- ψ : Heading of the ship
- V_{WR} : Relative wind speed
- ψ_{WR} : Relative wind direction relative to the bow, ship fixed; 0 means head winds
- V_{WT} : True wind speed
- B_{WT} : True wind angle in earth system
- α : Angle between ship heading and wave direction relative to the bow; 0 means head waves

The wind direction is defined as the direction from which the wind is coming and the wave direction is defined as the direction relative to the ship's heading from which the wave fronts are approaching.

2.2.2 ITTC-Wind

The air resistance will increase in line with how much of the hull is above waterline, and the relative velocity between the vessel and wind.

According to ITTC recommended guidelines [1], the wind-induced resistance could be calculated by:

$$R_{wind} = \frac{1}{2} \rho_{air} C_{DA} (\psi_{WRref}) A_{XV} V_{WRref}^2 - \frac{1}{2} \rho_A C_{DA}(0) A_{XV} V_G^2 \quad (2.6)$$

- A_{XV} : Ship's area of maximum transverse section exposed to the wind
- C_{DA} : Wind resistance coefficient
- ρ_{air} : Mass density of air
- V_{WRref} : Relative wind speed at reference height

- ψ_{WRref} : Relative wind direction at reference height; 0 means heading wind

Data sets of the wind resistance coefficient C_x depends on ship types and the relationship between C_x and C_{DA} is:

$$C_x = -C_{DA}$$

- C_x : Data sets of the wind resistance coefficient

So, Eq.(2.6) could be rewritten as:

$$R_{wind} = -\frac{1}{2}\rho_A C_x(\psi_{\text{WRref}}) A_{XV} V_{\text{WRref}}^2 + \frac{1}{2}\rho_A C_x(0) A_{XV} V_G^2 \quad (2.7)$$

The relative wind velocity at the reference height V_{WRref} in Eq.(2.7) could be given by:

$$V_{\text{WRref}} = \sqrt{V_{\text{WTref}}^2 + V_G^2 + 2V_{\text{WTref}}V_G \cos(\psi_{\text{WT}} - \psi)}$$

The relative wind direction at the reference height ψ_{WRref} in Eq.(2.7) could be given by:

$$\begin{aligned} \text{If: } V_G + V_{\text{WTref}} \cos(\psi_{\text{WT}} - \psi) &\geq 0 \\ \psi_{\text{WRref}} &= \tan^{-1} \frac{V_{\text{WTref}} \sin(\psi_{\text{WT}} - \psi)}{V_G + V_{\text{WTref}} \cos(\psi_{\text{WT}} - \psi)} \\ \text{If: } V_G + V_{\text{WTref}} \cos(\psi_{\text{WT}} - \psi) &< 0 \\ \psi_{\text{WRref}} &= \tan^{-1} \frac{V_{\text{WTref}} \sin(\psi_{\text{WT}} - \psi)}{V_G + V_{\text{WTref}} \cos(\psi_{\text{WT}} - \psi)} + 180^\circ \end{aligned}$$

2.2.3 Hollenbach

To predict the resistance in the calm water, the velocity and main dimensions of vessel should be taken into account. There are many methods for calculation and Hollenbach is the most recent empirical method for commercial vessels, which could estimate viscous resistance, friction resistance and wave resistance. This method is based on regression analysis of 433 ship models, and its calculation deeply depends on the vessels main dimensions [2].

Hollenbach method should be start with introducing the total resistance R_{Tmean} caused by the reaction between hull and calm water, including viscous resistance, friction resistance and wave resistance. The viscous resistance and wave resistance could be consider as the residual resistance. So, R_{Tmean} could be expressed as:

$$R_{Tmean} = R_{Fm} + R_R = \frac{\rho_{sea}}{2} V_G^2 \cdot (C_{Fm} \cdot S + C_R \cdot B \cdot T) \quad (2.8)$$

- R_{Tmean} : Mean total resistance, which caused by the reaction between hull and calm water
- R_{Fm} : (mean) Friction resistance
- R_R : Residual resistance

- ρ_{sea} : Mass density of sea water
- C_{Fm} : (mean) Friction resistance coefficient
- S : Wetted surface of ship
- C_R : Residual resistance coefficient
- B : Beam of ship
- T : Draft of ship

The (mean) friction resistance coefficient in Eq.(2.8) can be expressed as:

$$C_F = \frac{0.075}{[\log(R_e) - 2]^2} \quad (2.9)$$

- R_e : Reynold's number

Where:

$$R_e = \frac{V_G \cdot L}{\nu}$$

- L : Vessel's length between perpendiculars
- ν : Viscosity of sea water

By Hollenbach method, the residual coefficient C_R in Eq.(2.8) can be expressed as:

$$\begin{aligned} C_{R \text{ Hollenbach}} = & C_{R, \text{Standard}} \cdot C_{R, FnKrit} \cdot k_L \\ & \cdot \left(\frac{T}{B}\right)^{a1} \cdot \left(\frac{B}{L}\right)^{a2} \cdot \left(\frac{L_{OS}}{L_{wl}}\right)^{a3} \cdot \left(\frac{L_{wl}}{L}\right)^{a4} \\ & \cdot \left(\frac{D_p}{T_A}\right)^{a6} \cdot \left[1 - \frac{T_A - T_F}{L}\right]^{a5} \cdot (1 - N_{Rud})^{a7} \\ & \cdot (1 - N_{Brac})^{a8} \cdot (1 - N_{Boss})^{a9} \cdot (1 - N_{Thr})^{a10} \end{aligned} \quad (2.10)$$

- L_{wl} : Length of water line
- T_A : Draft of aft propeller
- T_F : Draft of fore propeller
- D_P : Propeller diameter
- N_{Rud} : Number of rudders
- N_{Brac} : Number of brackets
- N_{Boss} : Number of bossings
- N_{Thr} : Number of side thrusters

$C_{R, \text{Standard}}$ in Eq.(2.10) is defined as:

$$\begin{aligned} C_{R, \text{Standard}} = & b_{11} + b_{13} \cdot F_n + b_{13} \cdot F_n^2 + C_B \cdot (b_{21} + b_{22} \cdot F_n + b_{23} \cdot F_n^2) \\ & + C_B^2 \cdot (b_{31} + b_{32} \cdot F_n + b_{33} \cdot F_n^2) \end{aligned} \quad (2.11)$$

- C_B : Block coefficient

Where:

$$C_B = \frac{\Delta}{L \cdot B \cdot T}$$

- Δ : Displacement, and only in above equation Δ represents displacement

$$F_n = \frac{V_G}{\sqrt{g \cdot L_{fn}}}$$

- F_n : Froudes number
- L_{fn} : Froudes length
- g : Gravitational acceleration

However, the Froudes length is based on the value of the relation L_{OS}/L , where vessel's length over surface L_{OS} . According to Oosterveld's work [3], L_{OS} , which is dependent on the loading condition, is defined as:

- for design draught, is it the length between aft end of design waterline and the most forward point at the vessel below the design waterline.
- for ballast draught is it the length between aft end of the design and the forward end of the hull at ballast waterline.

The relation between L_{OS}/L and L_{fn} could be shown in following Table 2.1:

Table 2.1: Definition of Froudes length: L_{fn}

Values of L_{OS}/L	Values of L_{fn}
$L_{OS}/L < 1.0$	L_{OS}
$1.0 < L_{OS}/L < 1.1$	$L + 2/3 \cdot (L_{OS} - L)$
$L_{OS}/L > 1.1$	$1.0667 \cdot L$

$C_{R,FnKrit}$ in Eq.(2.10) is defined as:

$$C_{R,FnKrit} = \max \left[1.0, \left(\frac{F_n}{F_{n,krit}} \right)^{c_1} \right] \quad (2.12)$$

$F_{n,krit}$ in above equation is:

$$F_{n,krit} = d_1 + d_2 \cdot C_B + d_3 \cdot F_n^2 \quad (2.13)$$

k_L in Eq.(2.10) is defined as:

$$k_L = e_1 \cdot L^{e_2} \quad (2.14)$$

These formulas are valid when Froudes number in following intervals:

$$\begin{aligned} F_{n,\min} &= \min(f_1, f_1 + f_2(f_3 - C_B)) \\ F_{n,\max} &= g_1 + g_2 C_B + g_3 C_B^3 \end{aligned} \quad (2.15)$$

If $F_n > F_{n,\max}$, the maximum resistance would be:

$$R_{Tmax} = h_1 \cdot R_{Tmean} \quad (2.16)$$

If $F_n < F_{n,\min}$, $C_{R,FnKrit}$ and k_L in Eq.(2.10) should be set to 1.0 for calculating the minimum resistance R_{Tmin} .

Coefficients $a \sim h$ in equations above: Eq.(2.10), Eq.(2.11), Eq.(2.12), Eq.(2.13), Eq.(2.14) and Eq.(2.15), could be found in Molland's work [4].

In this project, R_{hull} could be expressed as:

$$R_{hull} = \begin{cases} R_{Tmin} & F_n < F_{n,\min} \\ R_{Tmean} & F_{n,\min} \leq F_n \leq F_{n,\max} \\ R_{Tmin} & F_n > F_{n,\max} \end{cases} \quad (2.17)$$

2.2.4 STAwave-1

Parameters of wave and main dimensions of vessel should be taken into account, when we calculate the resistance caused by head waves. STAwave-1 is a correction method to estimate the added resistance in waves with limited input data. As we stated, this method can be applied when a ship has limited pitch and heave. According to Boom's work [5], this method is valid when the incoming wave are in the bow sector, within ± 45 degrees off the bow. Its formula is:

$$R_{wave} = \frac{1}{16} \rho_{sea} g H_{W1/3}^2 B \sqrt{\frac{B}{L_{BWL}}} \quad (2.18)$$

- $H_{W1/3}$: Significant height of waves
- L_{BWL} : Length of the bow on the water line to 95% of maximum beam as shown in Figure 2.2

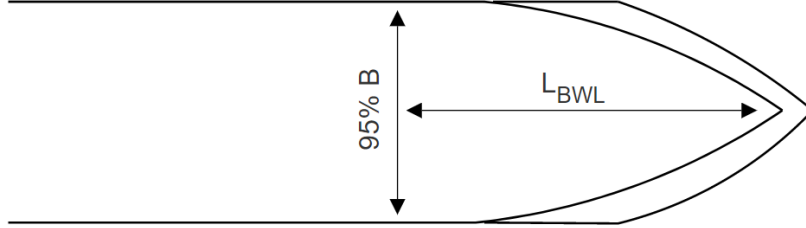


Figure 2.2: Definition of L_{BWL}

2.3 Path planning

Path planning, also named as motion planning, usually aims to find the shortest path, the path with lowest cost or the path who would take shortest time. One of the most popular methods to solve this shortest path problem is the Dijkstra algorithm, which is used in this project and the objective is to find the optimal route causing minimum fuel consumption.

2.3.1 Dijkstra

Dijkstra's algorithm, was conceived by the Dutch computer scientist Edsger Dijkstra [6]. Its basic thought is the process what from the start and gradually expand. In the process of exploring, record the path each to a point, and call the label.

The original Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step. However, in this project, the ship will move in the grids, and the distance between each coordinate point is equal (an assumption in Chapter 3) but the cost of going through are different. So, in this case, the node at the beginning of the path could be called as the origin node, and the total cost to go to the node Y could be defined as the cost from the origin node to Y. Then, the Dijkstra's algorithm would improve them step by step:

- step 1 Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
- step 2 Assign to every node a cost value based on resistance. Set the initial node as current.
- step 3 For the current node, consider all of its unvisited neighbours and calculate their cost through the current node. Compare the newly calculated cost to the current assigned value and assign the lower one.
- step 4 When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
- step 5 Stop if the destination node has been marked visited. The algorithm has finished.
- step 6 Otherwise, select the unvisited node that is marked with the lowest cost, set it as the new "current node", and go back to step 3.

Based on these steps, a pseudocode algorithm could be shown as:

```
1  create US,VS
2  while US is not empty:
3      pick u_min in VS as u
4      for each unvisited neighbour v of u:
5          CC(v) = cost(u, v)
6          if CC(v) < min cost(u, v):
7              min cost(0,v) = CC(v)
8      Ncost(v_min)=Ncost(u)+cost(u, v_min)
9      if v_min == destination node:
10         find EP
11     else:
12         removed v_min from US
13         add v_min into VS
```

Listing 2.1: Pseudocode Dijkstra algorithm

- *US*: Unvisited set
- *VS*: Visited set

- $Ncost(u)$: Cost of the efficient path to node u
- u_{min} : Node who has min $Ncost(u)$
- u : Current node
- $cost(a, b)$: Cost of the path from node a to node b
- v : Neighbour-nodes of u
- $CC(v)$: Current cost of the path from the original node to v if it were to go through u
- $path(a, b)$: Current path from node a to node b
- v_{min} : Neighbour-nodes of u , who has min $cost(u, v)$
- EP : Path to node destination node, whose cost equals to $Ncost(v_{min})$

Chapter 3

Programming process

In this chapter, the programming language and usage of packages would be introduced briefly. Then, some assumptions would be stated, which have a strong influence on modelling and processing. The programming process would be elaborated in final.

3.1 Python

In this project, Python, which is an interpreted high-level general-purpose programming language, is used to analyze data and find the optimum path.

The benefits of choosing Python for this project are:

- a) Python is versatile, readable, well-structured and easy to use and read
- b) Extensive support libraries make it easy to plot and handle data

However, disadvantages are linked with these benefits:

- a) Compared to C/C++, Python has a lower compiling speed and would cause a higher memory consumption
- b) Compared to MATLAB, Python is not convenient in matrix data processing and some mathematical calculations

3.1.1 Packages

Modules in Python could be simply considered as Python files, which has a specific functionality, and packages are a way of structuring Python's module namespace by using 'dotted module names'. The use of dotted module names saves the authors of multi-module packages from having to worry about each other's module names. Moreover, these standard library packages provide methods to accomplish some tasks.

Standard library packages used in this project are shown below:

- a) **Math**: provides access to the mathematical functions
- b) **NumPy**: uses for working with arrays and matrices.
- c) **Pandas**: provides open source data analysis and manipulation tools
- d) **Matplotlib**: creates static, animated and interactive visualizations

3.2 Assumptions

For modelling and programming, there are some assumptions and limitations used for reasonable simplification:

- (1) The ship would drive in grid with boundaries, from the origin node to the destination node. Then in this grid, the north is regarded as the positive direction of the y-axis and the east is regarded as the positive direction of the x-axis
- (2) Each node in the grid, except those on the boundary, has only four neighbour-nodes: North node, East node, South node and West node
- (3) The distances from the current node to its any neighbour-nodes are same
- (4) The ship could only travel from the current node to its neighbour-node
- (5) The ship would keep constant velocity in the whole voyage
- (6) The ship dimensions, like draft, would be constant in the whole voyage
- (7) The ship could only go in four directions: North, East, South and West
- (8) The ship could only change course at nodes
- (9) Changing course would not cause fuel consumption
- (10) Wind speeds and directions at different heights, like reference height and the vertical position of the anemometer, are same
- (11) Type of this ship is considered as the general cargo
- (12) The ship is single-screw and fully loaded, which means its draught keep at design value
- (13) There are no obstacles in grid
- (14) Ships can drive along borders, but it could not cross borders

3.3 Programming process

In this section, the algorithm of solving this project would be given. Details about modelling and programming would be discussed in subsections, and codes would be attached in appendix.

Based on these assumptions and ocean meteorological data, we could model this problem. The algorithm of solution could be expressed step by step:

- step 1 Data processing: read data and reconstruct data
- step 2 Cost calculation: for each velocity, calculate cost for each node and each direction. In other words, for a given velocity, there are 4 types of cost on each node.
- step 3 Path finding: for each velocity, find an effective path. Record the total cost of those velocities, under which the ship could arrive the destination

step 4 Velocity picking: compare different efficient paths of various velocities, pick the velocity which caused minimum cost

step 5 Failure and solution: If the ship could not complete this task under any velocity, more iterations are needed and go back to step 2

The pseudocode algorithm of this project could be shown as:

```

1 read data and reconstruct data
2
3 for i in iterations:
4     calculate velocity
5     for each velocity:
6         for each node:
7             for each direction:
8                 calculate cost
9
10        find effective path
11        if time <= given time:
12            record total cost
13        else:
14            could not complete task
15
16 if no velocity is eligible:
17     ask for more iteration
18 else:
19     compare total cost of different paths
20     get efficient velocity

```

Listing 3.1: Pseudocode algorithm of project

3.3.1 Data processing

In this part, the data of ocean weather would be read and reconstructed as matrices, which are convenient for further calculation.

The initial data of wind is comma-separated values (CSV) files. This kind of delimited text file would use some methods to separate data. In this project, the wind data is split by semicolons and rows. The function for dealing with these files could be found in Listing A.1.

The initial data of wave is saved as common Excel files, where data would be separated by rows and columns. So, it would be little different in handling files, which could be found in Listing A.2.

3.3.2 Cost calculation

Based on Assumption (3), (4) and (5), the time for this ship to travel from any node to its neighbour node would be same, we could define it as Δt . Then, based on Eq.(2.1) and Eq.(2.2), the cost between two neighbour nodes a and b could be expressed as:

$$\begin{aligned}
 \Delta C(a, b) &= \Delta P(a, b) \cdot \Delta t \\
 &= \Delta R(a, b) \cdot V_G \cdot \Delta t
 \end{aligned} \tag{3.1}$$

Although changing course would not increase cost, the heading angle of ship will still affect the resistance caused by wind and wave, in which case resistance needs to be calculated for each direction. This is the reason why limitations

are put on directions. Owing to Assumption (2) and (7), we could map the ship heading angle to 4 values: 0, 90, 270 and 360 degrees for representing that ship would drive from current nodes to North neighbour nodes, East neighbour nodes, South neighbour nodes and West neighbour nodes respectively. Then, according to Eq.(2.5) and Eq.(3.1), the cost function would be separated to 4 parts by ship motions. For example, if this ship would travel from the current node a to its north neighbour node c , the cost would be:

$$\begin{aligned}\Delta C_{North}(a, c) &= \Delta P_{North}(a, c) \cdot \Delta t \\ &= \Delta R_{North}(a, c) \cdot V_G \cdot \Delta t \\ &= [\Delta R_{wind, North}(a, c) + \Delta R_{wave, North}(a, c) \\ &\quad + \Delta R_{hull}(a, c)] \cdot V_G \cdot \Delta t\end{aligned}\quad (3.2)$$

$\Delta C_{East}(a, d)$, $\Delta C_{South}(a, e)$ and $\Delta C_{West}(a, f)$ could be calculated by similar methods.

Cost caused by wind

This part is discussed with Ziwen Wang (Student number:546844).

According to ITTC [1], based on Assumption (11), data sets of the wind resistance coefficient C_x could be determined:

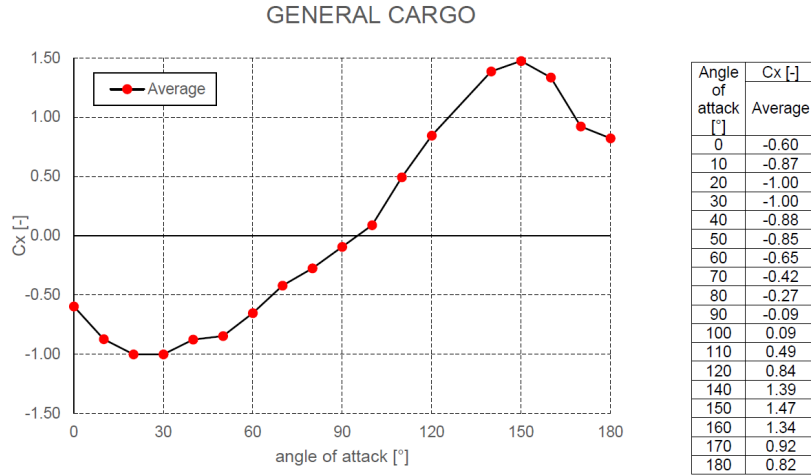


Figure 3.1: Data sets of the wind resistance coefficient C_x for general cargos

For convenience, $C_x(130^\circ)$ would be calculated, and C_x would be extended to 360 degrees while each element in $C_{x_extended}$ be changed to its own opposite value. Then, based on Assumption (10), Eq.(2.7) would be rewritten as:

$$R_{wind} = \frac{1}{2} \rho_A C_{x_extended}(\psi_{WR}) A_{XV} V_{WR}^2 - \frac{1}{2} \rho_A C_{x_extended}(0) A_{XV} V_G^2 \quad (3.3)$$

where:

$$V_{WR} = \sqrt{V_{WT}^2 + V_G^2 + 2V_{WT}V_G \cos(\psi_{WT} - \psi)}$$

$$\begin{aligned}
&\text{If: } V_G + V_{WT} \cos(\psi_{WT} - \psi) \geq 0 \\
&\quad \psi_{WR} = \tan^{-1} \frac{V_{WT} \sin(\psi_{WT} - \psi)}{V_G + V_{WT} \cos(\psi_{WT} - \psi)} \\
&\quad \text{Range: } -90^\circ \sim 90^\circ \\
&\text{If: } V_G + V_{WT} \cos(\psi_{WT} - \psi) < 0 \\
&\quad \psi_{WR} = \tan^{-1} \frac{V_{WT} \sin(\psi_{WT} - \psi)}{V_G + V_{WT} \cos(\psi_{WT} - \psi)} + 180^\circ \\
&\quad \text{Range: } -90^\circ \sim 90^\circ \quad \text{and} \quad 90^\circ \sim 270^\circ \\
&\text{Then, if: } \psi_{WR} < 0 \\
&\quad \psi_{WR} = \psi_{WR} + 360^\circ \\
&\quad \text{Range: } 0^\circ \sim 90^\circ \quad \text{and} \quad 90^\circ \sim 270^\circ \quad \text{and} \quad 270^\circ \sim 360^\circ
\end{aligned}$$

After getting R_{wind} , we could calculate cost caused by wind. The (part of) function for calculating cost caused by wind could be found in Listing A.3.

Cost caused by wave

Cost caused by wave is similar than that caused by wind, which needs to be separated as 4 parts. According to Eq.(2.18) and based on Assumption (7), this function could be found in Listing A.4.

Cost caused by the reaction between hull and calm water

According to Molland's work [4], based on Assumption (12), coefficients $a \sim h$ would be shown as below:

Table 3.1: Coefficients of Hollenbach method

a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	
-0.3382	0.8086	-6.0258	-3.5632	9.4405	0.0146	0	0	0	0	
b11	b12	b13			d1	d2	d3			
-0.57424	13.3893	90.596			0.854	-1.228	0.497			
b21	b22	b23			e1	e2				
4.6614	-39.721	-351.483			2.1701	-0.1602				
b31	b32	b33			f1	f2	f3			
-1.14215	-12.3296	459.254			0.17	0.2	0.6			
g1	g2	g3			h1					
0.642	-0.635	0.15			1.204					

In addition, a matlab scrip has been given in blackboard, which has been rewritten as a Python script (Listing A.5).

Based on this script and according to Eq.(2.8) and Eq.(2.17), the above script would be modified. The modified version could be found in Listing A.6.

3.3.3 Path finding

This part is discussed with Ziwen Wang (Student number:546844).

For better display, this work for path finding is modified based on an open source project written by Atsushi, which aims for a robot to plan a path rounding

obstacles.

In Atsushi's project, there are some obstacles and robot has its own radius. So, this robot could move in 8 different directions and find the shortest way to round these obstacles and arrive the goal point by Dijkstra algorithm. In this case, the cost between any two neighbour nodes is set to the same. Here is an example below (result of Listing A.7):

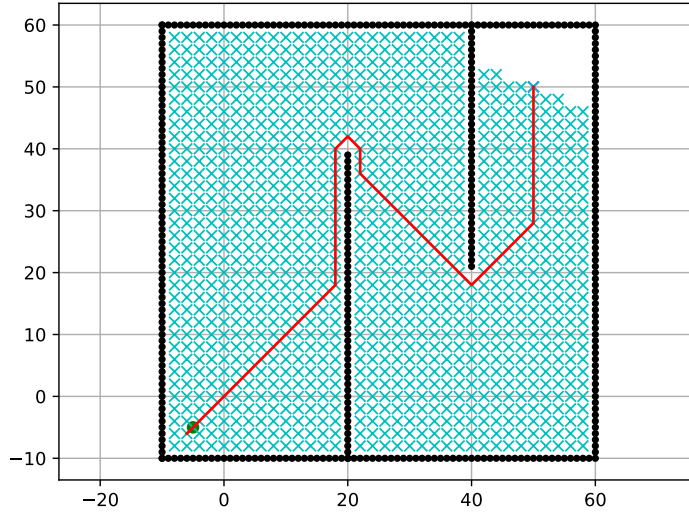


Figure 3.2: Grid based Dijkstra planning (Atsushi's project)

However, some of these principles in Atsushi's project are against previous assumptions, in which case some modifications are necessary.

Based on Assumption (4) and (7), the number of movement directions would be deleted to four. Owing to Assumption (13), all obstacles are removed. The robot radius, which could be considered as the beam of ship, would be discarded as well because of Assumption (14). This removal just eliminates the influence of beam on ship movement to ensure that this ship could move on all nodes in grid, even those nodes on the boundaries, but it does not mean that the effect of beam on resistance would be changed. In final, owing to the influence of ship heading angle, cost of movements between two nodes has an independent value for each direction. So, to find the path whose cost is minimum in a given velocity, the cost of each direction on every node would be replaced by calculated results in above sections. After finding efficient path, the cost of this path could be given. According to Eq.(3.1), total cost of each path could be expressed as:

$$C_{total}(V_{G,j}) = \sum_{n=1}^N \Delta C_n(V_{G,j}) \quad (3.4)$$

- $C_{total}(V_{G,j})$: Total cost of the voyage for the given velocity $V_{G,j}$
- ΔC_n : Cost per section of this path, ΔC_1 means the cost from the start

point to the first driving point, ΔC_1 means the cost from the last point before the end to the goal point

The modified version could be found in Listing A.8.

3.3.4 Velocity picking

The minimum velocity could be defined as:

$$V_{min} = \frac{l_{min}}{t} = \frac{(|sx - gx| + |sy - gy|) * dist}{t} \quad (3.5)$$

- V_{min} : Minimum velocity
- l_{min} : Length of shortest path
- sx : x position of the start point
- gx : x position of the goal point
- sy : y position of the start point
- gy : y position of the goal point
- $dist$: distance between two neighbour nodes

The maximum velocity could be defined as:

$$V_{max} = \frac{l_{max}}{t} = \frac{(xcol \cdot yrow) * dist}{t} \quad (3.6)$$

- V_{max} : Maximum velocity
- l_{max} : Length of longest path
- $xcol$: Number of columns in grid
- $yrow$: Number of rows in grid

However, if each velocity would be calculated in program, it will cause computing time too long. So, setting a iteration limitation would be a good idea to improve efficiency.

After comparing total costs of efficient paths under different velocities, we could choose the optimal speed. Codes for this function could be found in Listing A.9.

Chapter 4

Modelling

4.1 Parameters for testing

For testing, parameters used in testing codes (Listing A.10), could be assumed as Table 4.1:

Table 4.1: Parameters for testing

Parameters	Values	Units	Meanings
Variable			
time	0.1;1;10;100	h	given time
num_iter	30;50		limitation of iterations
Grid			
sx	3		x position of the start point
sy	3		y position of the start point
gx	47		x position of the goal point
gy	47		y position of the goal point
xcol	50		number of columns
yrow	50		number of rows
dist	1000	m	distance between neighbour nodes
Ship			
L	200	m	length of this ship
Lwl	195	m	length of water line
Lbwl	38	m	length of the bow on the water line to 95% of maximum beam
Los	196.5	m	length over surface
B	33	m	beam
TF	11.6	m	draft of fore propeller
TA	11.4	m	draft of aft propeller
CB	0.855		block coefficient
S	10500	m^2	wetted surface
Axv	1600	m^2	area of maximum transverse section exposed to the wind
Dp	7	m	propeller diameter
NRud	1		nubmer of rudder
NBrac	1		number of brackets

Parameters	Values	Units	Meanings
NBoss	1		number of bossings
NThr	1		nubmer of side thrusters
Constant			
rho_air	1.293	kg/m^3	density of air
rho_sea	1025	kg/m^3	density of sea
nu_sea	1.1395E-06	m^2/s	viscosity of sea
g	9.81	m/s^2	gravitational acceleration

4.2 Tests and results

The aim of this project is to find the optimum velocity and the efficient path under this speed, which would lead to minimum cost. So, for this aim, these tasks are set as that a ship needs to drive from the start point (3,3) to the goal point (47,47) within a given time and number of iterations. Variables for these 5 tests are shown as:

Table 4.2: Tests and results:

Test	Given time[h]	Number of iterations	results
1	0.1	30	success
2	1	30	success
3	10	30	success
4	100	30	failure
5	100	50	success

Table 4.3: Results of successful tests:

Test	Optimal Velocity[knot]	Minimum Total Cost[MJ]	Paths
1	480.603	$2.7 * 10^{62}$	Figure 4.1
2	48.600	$5.4 * 10^8$	Figure 4.2
3	5.454	97287.277	Figure 4.3
5	0.648	9348.747	Figure 4.4

Results of test 1 is shown in Table 4.3 and Figure 4.1, the ship heading east firstly. This is because excessive speed would make the influence of heading angles greater, in which case all costs caused by eastward movements are cheaper than that of northward movements. Results of test 2 is shown in Table 4.3 and Figure 4.2, where the path of test 2 is similar than that of first task.

Owing to the impact of moving too fast, the cost between two neighbour nodes is so big that the ship would prefer to go straight to the target point like the first two tests. The optimum path causing minimum costs would always be found in shortest paths.

As is shown in the Table 4.3, the smaller the ship speed is, the lower the cost

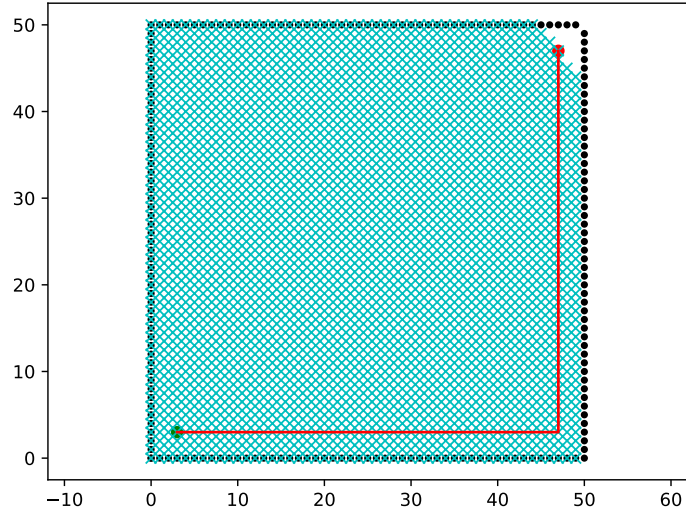


Figure 4.1: Result of Test 1

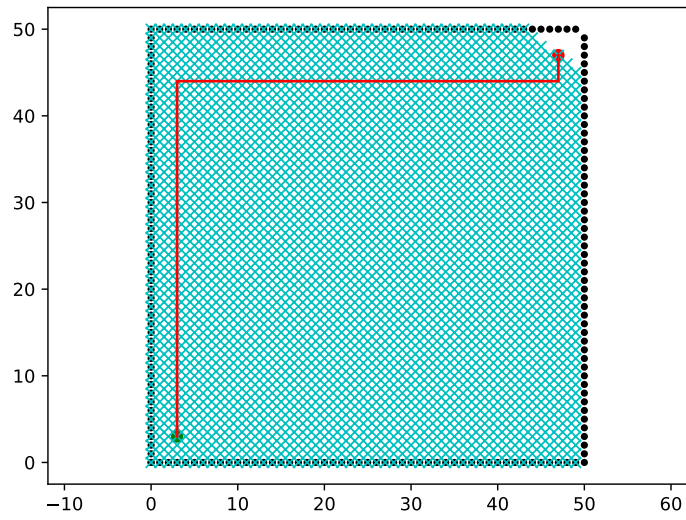


Figure 4.2: Result of Test 2

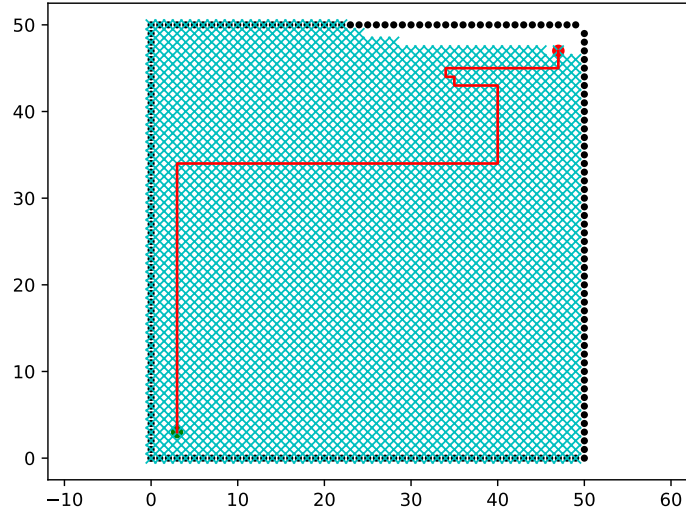


Figure 4.3: Result of Test 3

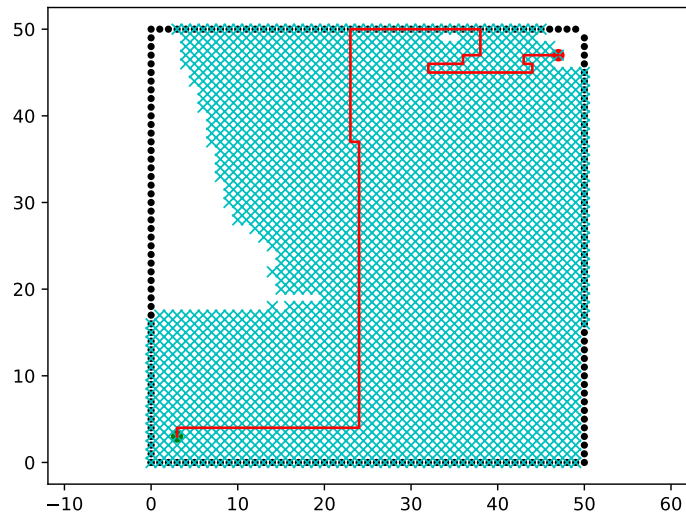


Figure 4.4: Result of Test 5

between neighbour nodes would be, so is the total cost. If the cost of more steps would be cheaper, shortest paths with high costs would be given up, which could be proved by results of 3rd test and 5th test (Figure 4.3 and Figure 4.4). On the other side, for each velocity, the increasing path length need take more time, which may make this task not finished in given time. Moreover, the algorithm iterates from the minimum speed, and when following the shortest path, the lower limit speed for completing the task on time is the smallest. This low limit is increasing with the growing path length. So if the starting velocity is low enough to avoid the shortest path, it would need more iterations to find velocity which reach the new lower speed limit. That is why test 4 fails and test 5 succeeds with more iterations.

As it states above, the incresement of cost is closely related with the growth of speed and the algorithm iterates from the minimum speed. So, normally, the optimal velocities would be the minimum velocity which reach the lower speed limit, which is true for first 3 tests. However, when the speed is low, the ocean weather would play a more important role in resistance generation. So, the minimum velocity may not be the optimum velocity any more, which is proved in test 5. The optimum velocity in test 5 is the second minimal velocity under which this ship could arrive the goal point with in the given time.

Chapter 5

Conclusion

5.1 Conclusion

In this project, there is a ship that needs to go to the designated place within the specified time. In this case, an algorithm to find the optimum velocity and path was designed for the minimal consumption energy. In this algorithm, costs would be calculated by ITTC recommended method, Hollenbach method and STAwave-1method while paths would be found by Dijkstra planning.

It is successful to solve two interrelated questions, power and time, by controlling velocity firstly. After comparing and checking, the optimal velocity would be given in the terminal and the efficient path would be shown by plotting.

5.2 Reflection and future work

After reflecting on the whole project, there are quite a few aspects which could be improved in the future work.

- In this project, the cost are calculated directly from the effective power, which is different from the power of engine. The conversion efficiency between these two powers is relative with ship velocity as well, which should be taken into account in real optimal velocity calculation.
- There are some corrections in equations for calculating resistance. These corrections would be implemented for better results.
- This project are based on many assumptions (section 3.2), which could be improved as well. For example, in Assumption (9), changing course would not cause fuel consumption, which could be changed to reduce numbers of turning.

Bibliography

- [1] ITTC Recommended Guidelines: Preparation, Conduct and Analysis of Speed/Power Trials (7.5-04-01-01.1). 2017.
- [2] U. Hollenbach. Estimating resistance and propulsion for single-screw and twin-screw ships. 1998.
- [3] Marinus Willem Cornelis Oosterveld and Peter van Oossanen. Further computer-analyzed data of the wageningen b-screw series. *International shipbuilding progress*, 22(251):251–262, 1975.
- [4] Dominic a Hudson Anthony F Molland, Stephen R Turnock. *Ship Resistance and Propulsion: Practical Estimation of Ship Propulsion Power*. Cambridge University Press, 2011.
- [5] Henk van den Boom, Hans Huisman, and Frits Mennen. New guidelines for speed/power trials. *Level playing field established for IMO EEDI*. *SWZ Maritime*, pages 1–11, 2013.
- [6] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

Appendix A

Codes

A.1 Functions

```
1 def ReadCsvWind(filepath,xcol,yrow):
2     """
3     Written by Weijian Yang, email: weijiany@stud.ntnu.no
4     Function: read data & reconstruct data
5     Parameters:
6         filepath: filepath of CSV files (delimiter is ";")
7         xcol: number of columns in grid
8         yrow: number of rows in grid
9     Return value:
10        dataname: matrix of reconstructed data
11    """
12    file = open(filepath, "rb")
13    filedata=np.loadtxt(file, delimiter=";")
14    file.close()
15    filearray=np.array(filedata)
16
17    """ +2: just in case for that: data is not enough for grid """
18    num_row=xcol+2
19    num_col=yrow+2
20    dataname = np.zeros(shape=(num_row,num_col))
21    for i in range(num_row):
22        for j in range(num_col):
23            dataname[i,j]=filearray[i, j]
24    return dataname
```

Listing A.1: Function for reading and reconstructing wind data (subsection 3.3.1)

```
1 def ReadCsvWave(filepath,xcol,yrow):
2     """
3     Written by Weijian Yang, email: weijiany@stud.ntnu.no
4     Function: read data & reconstruct data
5     Parameters:
6         filepath: filepath of normal Excel files or CSV files where data is
7         seperated by rows and columns
8         xcol: number of columns in grid
9         yrow: number of rows in grid
10    Return value:
11        dataname: matrix of reconstructed data
12    """
```

```

12 file =pd.read_csv(filepath)
13
14 """ +2: just in case for that: data is not enough for grid """
15 num_row=xcol+2
16 num_col=yrow+2
17 dataname = file.values[0:num_row,0:num_col]
18 return dataname

```

Listing A.2: Function for reading and reconstructing wave data (subsection 3.3.1)

```

1 def WindResCosFun(Vs,Axv,rho_air,wind_u,wind_v,Cair_extend,dt):
2     """
3     Written by Weijian Yang, email: weijiany@stud.ntnu.no
4     Function: calculate cost caused by wind
5     Method: ITTC recommended method (2017)
6     Parameters:
7         Vs: measured ship's speed over ground
8         Axv: ship's area of maximum transverse section exposed to the wind
9         rho_air: mass density of air
10        wind_u: wind speed (north-south direction)
11        wind_v: wind speed (east-west direction)
12        Cair_extend: extended data sets of the wind resistance coefficient (0~360
13        degrees)
14        dt: time needed for travelling between two neighbour nodes
15    Return value:
16        CF_Wind_North: cost caused by wind, if travelling to north neighbour
17        nodes
18        CF_Wind_East: cost caused by wind, if travelling to east neighbour nodes
19        CF_Wind_South: cost caused by wind, if travelling to south neighbour
20        nodes
21        CF_Wind_West: cost caused by wind, if travelling to west neighbour nodes
22    """
23    angle_ship=np.mat([0,90,180,270]) #Angle of ship: North, East, South and
24    West
25    for k in range(angle_ship.shape[1]):
26        vel_wind_squire=np.mat(np.zeros(wind_v.shape))
27        vel_wt_squire=np.mat(np.zeros(wind_u.shape))
28        angle_wt=np.mat(np.zeros(wind_u.shape))
29        angle_wt_rel=np.mat(np.zeros(wind_v.shape))
30        angle_rel=np.mat(np.zeros(wind_u.shape))
31        Cx=np.mat(np.zeros(wind_v.shape))
32        if k == 0:
33            wind_u_north=wind_u+Vs
34            for i in range(wind_u.shape[0]):
35                for j in range(wind_v.shape[1]):
36                    vel_wt_squire[i,j]=wind_u_north[i, j]**2+wind_v[i, j]**2 #Vwt:
37                    true wind velocity
38                    if vel_wt_squire[i,j]==0:
39                        angle_wt[i,j]=0
40                    else:
41                        angle_wt[i,j]=math.degrees(math.acos(wind_v[i,j]/math.
42                        sqrt(vel_wt_squire[i,j]))) #Awt:true wind direction
43
44                    angle_wt_rel[i,j]=angle_wt[i,j]-angle_ship[0,k]+180 #Awt-
45                    Aship
46                    vel_wind_squire[i,j]=vel_wt_squire[i,j]+Vs**2+math.sqrt(
47                    vel_wt_squire[i,j])*Vs*math.cos(math.radians(angle_wt_rel[i,j])) #V_WRef
48                    nume=math.sqrt(vel_wt_squire[i,j])*math.sin(math.radians(
49                    angle_wt_rel[i,j]))
50                    deno=Vs+math.sqrt(vel_wt_squire[i,j])*math.cos(math.radians(
51                    angle_wt_rel[i,j]))

```

```

42         if deno==0:
43             if nume<0:
44                 angle_rel[i,j]==-90
45             if nume>0:
46                 angle_rel[i,j]=90
47             if nume==0:
48                 angle_rel[i,j]=0
49         elif deno<0:
50             angle_rel[i,j]=math.degrees(math.atan(nume/deno))+180 #
A_WRef
51         else:
52             angle_rel[i,j]=math.degrees(math.atan(nume/deno))
53             if angle_rel[i,j]< 0:
54                 angle_rel[i,j]=angle_rel[i,j]+360
55
56             weight=angle_rel[i,j]/10
57             index=int(angle_rel[i,j]//10)
58             Cx[i,j]=(weight-index)*Cair_extend[0,index+1]+(1-weight+index
59 )*Cair_extend[0,index]
60             Rwind_N=0.5*rho_air*Axv*(Cx*vel_wind_sqre-Cair_extend[0,0]*Vs**2)
61             CF_Wind_North=Rwind_N*Vs*dt
62
63         if k == 1:
64             wind_v_east=wind_v+Vs
65             for i in range(wind_u.shape[0]):
66                 for j in range(wind_v.shape[1]):
67                     vel_wt_sqre[i,j]=wind_u[i, j]**2+wind_v_east[i, j]**2 #Vwt:
true wind velocity
68                     if vel_wt_sqre[i,j]==0:
69                         angle_wt[i,j]=0
70                     else:
71                         """
72                         It is hard to shorten codes because equations for calculating
73                         true wind direction are different.
74                         For comparison: (north) angle_wt[i,j]=math.degrees(math.acos(
75                         wind_v[i,j]/math.sqrt(vel_wt_sqre[i,j])))
76                         """
77                         angle_wt[i,j]=math.degrees(math.asin(wind_v_east[i,j]/
78                         math.sqrt(vel_wt_sqre[i,j]))) #Awt:true wind direction
79
80         ...
81
82     return CF_Wind_North,CF_Wind_East,CF_Wind_South,CF_Wind_West

```

Listing A.3: Function for calculating cost caused by wind (parts of code)(section 3.3.2)

```

1 def WaveResCosFun(Vs,Lbwl,B,rho_sea,wave_d,wave_h,dt):
2     """
3     Written by Weijian Yang, email: weijiany@stud.ntnu.no
4     Function: calculate cost caused by wind
5     Method: STAwave-1
6     Parameters:
7         Vs: measured ship's speed over ground
8         Lbwl: length of the bow on the water line to 95\% of maximum beam
9         rho_sea: mass density of sea
10        wave_d: wave direction
11        wave_h: significant height of waves
12        dt: time needed for travelling between two neighbour nodes
13    Return value:
14        CF_Wave_N: cost caused by wave, if travelling to north neighbour nodes
15        CF_Wave_E: cost caused by wave, if travelling to east neighbour nodes

```

```

16         CF_Wave_S: cost caused by wave, if travelling to south neighbour nodes
17         CF_Wave_W: cost caused by wave, if travelling to west neighbour nodes
18     """
19     angle_ship=np.mat([0,90,180,270]) #Angle of ship: North, East, South and
    West
20     for k in range(angle_ship.shape[1]):
21         angle_rel=np.mat(np.zeros(wave_d.shape))
22         R_wave=np.mat(np.zeros(wave_h.shape))
23         for i in range(wave_d.shape[0]):
24             for j in range(wave_h.shape[1]):
25                 angle_rel[i,j]=wave_d[i,j]-angle_ship[0,k]
26                 while angle_rel[i,j]<-180 or angle_rel[i,j]>=180:
27                     if angle_rel[i,j]< -180:
28                         angle_rel[i,j]=angle_rel[i,j]+360
29                     if angle_rel[i,j]>=180:
30                         angle_rel[i,j]=angle_rel[i,j]-360
31
32                 if angle_rel[i,j]<-45 or angle_rel[i,j]>45:
33                     R_wave[i,j]=0
34                 else:
35                     R_wave[i,j]=rho_sea*g*B*math.sqrt(B/Lbwl)/16*wave_h[i,j]**2
36                 C_wave=R_wave*Vs*dt
37             if k==0:
38                 CF_Wave_N=C_wave
39             if k==1:
40                 CF_Wave_E=C_wave
41             if k==2:
42                 CF_Wave_S=C_wave
43             if k==3:
44                 CF_Wave_W=C_wave
45     return CF_Wave_N,CF_Wave_E,CF_Wave_S,CF_Wave_W

```

Listing A.4: Function for calculating cost caused by wave (section 3.3.2)

```

1     """
2     Given in class
3     Rewritten by Weijian Yang, email: weijiany@stud.ntnu.no
4     Function: calculate cost caused by the reaction between hull and calm water
5     Method: Hollenbach
6     """
7     #input some data
8     #Vsvec,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr
9     Vsvec=float(input('Velocity of Ships(m/s):'))
10    L=float(input('Length of Ship(meters):'))
11    Lwl=float(input('Length of Water Line(meters):'))
12    Los=float(input('Length over Surface(meters):'))
13    B=float(input('Beam(meters):'))
14    TF=float(input('Draft of Fore Propeller(meters):'))
15    TA=float(input('Draft of Aft Propeller(meters):'))
16    CB=float(input('Block coefficient:'))
17    S=float(input('Wetted Surface(square meters):'))
18    Dp=float(input('Propeller diameter(meters):'))
19    NRud=float(input('Number of rudders:'))
20    NBrac=float(input('Number of brackets:'))
21    NBoss=float(input('Number of bossings:'))
22    NThr=float(input('Number of side thrusters:'))
23
24
25    #Calculation of 'Froude length', Lfn:
26    if Los/L < 1:
27        Lfn = Los
28    elif (Los/L >= 1) & (Los/L < 1.1):
29        Lfn = L+2/3*(Los-L)

```

```

30 elif Los/L >= 1.1:
31     Lfn = 1.0667*L
32
33 # 'Mean' resistance coefficients
34 a = np.mat([-0.3382, 0.8086, -6.0258, -3.5632, 9.4405, 0.0146, 0, 0, 0, 0]) #a1
    means a[0,0]
35 b = np.mat([[[-0.57424, 13.3893, 90.5960],[4.6614, -39.721, -351.483],[-1.14215,
    -12.3296, 459.254]]) #b12 means b[0,1]
36 d = np.mat([0.854, -1.228, 0.497])
37 e = np.mat([2.1701, -0.1602])
38 f = np.mat([0.17, 0.20, 0.60])
39 g = np.mat([0.642, -0.635, 0.150])
40 # 'Minimum' resistance coefficients
41 a_min = np.mat([-0.3382, 0.8086, -6.0258, -3.5632, 0, 0, 0, 0, 0, 0])
42 b_min = np.mat
    ([[[-0.91424, 13.3893, 90.5960],[4.6614, -39.721, -351.483],[-1.14215, -12.3296, 459.254]])

43 d_min = np.mat([0, 0, 0])
44 e_min = np.mat([1, 0])
45 f_min = np.mat([0.17, 0.2, 0.6])
46 g_min = np.mat([0.614, -0.717, 0.261])
47
48 cc = 0
49 # Loop over velocities
50 for i in range(Vsvec.size):
51     Vs = Vsvec[0,i]
52
53     cc = cc + 1
54
55     Fn = Vs/((gravk*Lfn)**0.5) #Froude's number
56     Fnkrit_help0=np.mat([1,CB,CB**2]) # Build Matrix for using transpose:
        Fnkrit_help0.T
57     Fnkrit_help1 = d*Fnkrit_help0.T # Fnkrit_help1=[[x]] Matrix type
58     Fnkrit=Fnkrit_help1[0,0] # Fnkrit=x Float type
59     c1 = Fn/Fnkrit
60     c1_min = Fn/Fnkrit
61     Rns = Vs*L/nu #Reynold's number for ship
62     if Rns == 0: #Rns=0,log would get stuck
63         CFs =0
64     else :
65         CFs = 0.075/(math.log10(Rns)-2)**2 #ITTC friction line for ship
66
67
68 # Calculation of C_R for given ship
69 # Mean value
70 CRFnkrit = max(1.0, (Fn/Fnkrit)**c1)
71 kL = e[0,0]*L**(e[0,1])
72
73 # There is an error in the hollenbach paper and in Minsaas' 2003 textbook,
    which is corrected in this formula by dividing by 10
74 CRstandard_help0=np.mat([1,Fn,Fn**2])
75 CRstandard_help1=Fnkrit_help0*(b*CRstandard_help0.T)/10
76 CRstandard=CRstandard_help1[0,0]
77
78 #prod([T/B B/L Los/Lwl Lwl/L (1+(TA-TF)/L) Dp/TA (1+NRud) (1+NBrac) (1+NBoss)
    (1+NThr)].^a)
79 prod_help=np.mat([T/B,B/L,Los/Lwl,Lwl/L,1+(TA-TF)/L,Dp/TA,1+NRud,1+NBrac,1+
    NBoss,1+NThr])
80 prod_help1=np.mat(np.ones((1,10))) #build a Matrix[1,10]
81 for j in range(a.size): #prod_help=[prod_help[0,i].^a_min[0,i]]
82     prod_help1[0,j]=prod_help[0,j]**a[0,j]
83 prod_help2=np.prod(prod_help1,axis = 1) #prod function

```



```

84 prod=prod_help2[0,0]
85
86 CR_hollenbach = CRstandard*CRFnkrit*kL*prod
87 CR = CR_hollenbach*B*T/S #Resistance coefficient, scaled for wetted surface
88 C_Ts = CFs + CR #Total resistance coeff. ship
89 R_T_mean = C_Ts*rho/2*Vs**2*S #Total resistance to the ship
90
91 #Minimum values
92
93 #There is an error in the hollenbach paper and in Minsaas' 2003 textbook,
94 #which is corrected in this formula by dividing by 10
95 CRstandard_min_help = Fnkrit_help0*(b_min*CRstandard_help0.T)/10
96 CRstandard_min=CRstandard_min_help[0,0]
97 #prod([T/B B/L Los/Lw1/Lw1/L (1+(TA-TF)/L) Dp/TA (1+NRud) (1+NBrac) (1+NBoss)
98 # (1+NThr)].^a_min)
99 prod_help_min1=np.mat(np.ones((1,10)))
100 for j in range(a_min.size): #
101     prod_help=[prod_help[0,i].^a_min[0,i]]
102     prod_help_min1[0,j]=prod_help[0,j]**a_min[0,j]
103 prod_help_min2=np.prod(prod_help_min1,axis = 1) #prod
104 function
105 prod_min=prod_help_min2[0,0]
106
107 CR_hollenbach_min = CRstandard_min*prod_min
108 CR_min = CR_hollenbach_min*B*T/S
109
110 # Total resistance coefficient of the ship
111 C_Ts_min = CFs + CR_min
112 # Total resistance
113 R_T_min = C_Ts_min*rho/2*Vs**2*S
114 #Propulsion power
115 P_E_mean = R_T_mean * Vs #[W]
116 P_E_min = R_T_min * Vs #[W]
117
118 #print sth.
119 # print('Vs =', Vs/0.5144,'knots')
120 # print('Mean values')
121 # print('CRh:',CR)
122 # print('CF:',CFs)
123 # print('CT:',C_Ts)
124 # print('RT:',R_T_mean,'N')
125 # print('Minimum values')
126 # print('CRh:',CR_min)
127 # print('CF:',CFs)
128 # print('CT:',C_Ts_min)
129 # print('RT:',R_T_min)
130
131 # % Store results for plotting
132 CFsvec = np.mat(np.zeros((1,Vsvec.size)))
133 CRvec = np.mat(np.zeros((1,Vsvec.size)))
134 C_Tsvec = np.mat(np.zeros((1,Vsvec.size)))
135 R_T_meanvec = np.mat(np.zeros((1,Vsvec.size)))
136 CR_minvec = np.mat(np.zeros((1,Vsvec.size)))
137 C_Ts_minvec = np.mat(np.zeros((1,Vsvec.size)))
138 R_T_minvec = np.mat(np.zeros((1,Vsvec.size)))
139 P_E_meanvec = np.mat(np.zeros((1,Vsvec.size)))
140 P_E_minvec = np.mat(np.zeros((1,Vsvec.size)))
141 CFsvec[0,i] = CFs
142 CRvec[0,i] = CR
143 C_Tsvec[0,i] = C_Ts
144 R_T_meanvec[0,i] = R_T_mean

```

```

142 CR_minvec[0,i] = CR_min
143 C_Ts_minvec[0,i] = C_Ts_min
144 R_T_minvec[0,i] = R_T_min
145 P_E_meanvec[0,i] = P_E_mean
146 P_E_minvec[0,i] = P_E_min
147
148 #This is the returned matrix
149 resistancedata =np.hstack((Vsvec.T,R_T_meanvec.T,R_T_minvec.T,P_E_meanvec.T,
    P_E_minvec.T))

```

Listing A.5: Rewritten Python script based on the given MATLAB script (section 3.3.2)

```

1 def TotalResCosFun(Vs,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,rho_sea,
    nu_sea,g,dt):
2     """
3     Modified by Weijian Yang, email: weijiany@stud.ntnu.no
4     Function: calculate cost caused by the reaction between hull and calm water
5     Method: Hollenbach
6     Parameters:
7         Vs: measured ship's speed over ground
8         L: length between perpendiculars
9         Lwl: length of water line
10        Los: length over surface
11        B: beam
12        TA: draught of aft propeller
13        TF: draught of fore propeller
14        CB: block coefficient
15        S: wetted surface
16        Dp: propeller diameter
17        NRud: number of rudders
18        NBrac: number of brackets
19        NBoss: number of bossings
20        NThr: number of side thrusters
21        rho_sea: mass density of sea
22        nu_sea: mass viscosity of sea
23        g: gravitational acceleration
24        dt: time needed for travelling between two neighbour nodes
25    Return value:
26        C_T: cost caused by the reaction between hull and calm water
27    """
28    rho = 1025          #Density of sea water [kg/m^3]
29    gravk = 9.81        #Gravitational constant [m/s^2]
30    nu = 1.1395E-6      #Viscosity of sea water [m/s^2]
31
32    T = (TF+TA)/2
33    """ Calculation of 'Froude length', Lfn """
34    if Los/L < 1:
35        Lfn = Los
36    elif (Los/L >= 1) & (Los/L < 1.1):
37        Lfn = L+2/3*(Los-L)
38    elif Los/L >= 1.1:
39        Lfn = 1.0667*L
40
41    # 'Mean' resistance coefficients
42    a = np.mat([-0.3382, 0.8086, -6.0258, -3.5632, 9.4405, 0.0146, 0, 0, 0, 0]) #
    a1 means a[0,0]
43    b = np.mat([[ -0.57424, 13.3893, 90.5960],[4.6614, -39.721,
    -351.483],[ -1.14215, -12.3296, 459.254]]) #b12 means b[0,1]
44    d = np.mat([0.854, -1.228, 0.497])
45    e = np.mat([2.1701, -0.1602])
46    f = np.mat([0.17, 0.20, 0.60])

```

```

47 g = np.mat([0.642, -0.635, 0.150])
48
49 Fn = Vs/((gravk*Lfn)**0.5) #Froude's number
50 Fnkrit_help0=np.mat([1,CB,CB**2]) # Build Matrix for using transpose:
    Fnkrit_help0.T
51 Fnkrit_help1 = d*Fnkrit_help0.T # Fnkrit_help1=[[x]] Matrix type
52 Fnkrit=Fnkrit_help1[0,0] # Fnkrit=x Float type
53 c1 = Fn/Fnkrit
54 Rns = Vs*L/nu #Reynold's number for ship
55 if Rns == 0: #Rns=0,log would get stuck
56     CFs =0
57 else :
58     CFs = 0.075/(math.log10(Rns)-2)**2 #ITTC friction line for ship
59
60 """ Calculation of C_R for given ship """
61 # Mean value
62 CRFnkrit = max(1.0,(Fn/Fnkrit)**c1)
63 kL = e[0,0]*L**(e[0,1])
64
65 # There is an error in the hollenbach paper and in Minsaas' 2003 textbook,
    which is corrected in this formula by dividing by 10
66 CRstandard_help0=np.mat([1,Fn,Fn**2])
67 CRstandard_help1=Fnkrit_help0*(b*CRstandard_help0.T)/10
68 CRstandard=CRstandard_help1[0,0]
69
70 #prod([T/B B/L Los/Lwl Lwl/L (1+(TA-TF)/L) Dp/TA (1+NRud) (1+NBrac) (1+NBoss)
    (1+NThr)].^a)
71 prod_help=np.mat([T/B,B/L,Los/Lwl,Lwl/L,1+(TA-TF)/L,Dp/TA,1+NRud,1+NBrac,1+
    NBoss,1+NThr])
72 prod_help1=np.mat(np.ones((1,10))) #build a Matrix[1,10]
73 for j in range(a.size): #prod_help=[prod_help[0,i].^a_min[0,i]]
74     prod_help1[0,j]=prod_help[0,j]**a[0,j]
75 prod_help2=np.prod(prod_help1,axis = 1) #prod function
76 prod=prod_help2[0,0]
77
78 CR_hollenbach = CRstandard*CRFnkrit*kL*prod
79 CR = CR_hollenbach*B*T/S #Resistance coefficient, scaled for wetted surface
80 C_Ts = CFs + CR #Total resistance coeff. ship
81 R_T_mean = C_Ts*rho/2*Vs**2*S #Total resistance to the ship
82
83 Fn_min=min(f[0,0],f[0,0]+f[0,1]*(f[0,2]-CB))
84 Fn_max=g[0,0]+g[0,1]*CB+g[0,2]*CB**3
85
86 if Fn>Fn_max:
87     #R_T=h1*R_T_mean
88     R_T=1.204*R_T_mean #h1=1.204
89 elif Fn<Fn_min:
90     #CRFnkrit=kL=1.0
91     CR_min=CRstandard*prod*B*T/S
92     R_T= (CFs + CR_min)*rho/2*Vs**2*S
93 else:
94     R_T=R_T_mean
95
96 C_T = R_T*Vs*dt
97 return C_T

```

Listing A.6: Function for calculating cost caused by the reaction between hull and calm water(section 3.3.2)

```

1 """
2 Grid based Dijkstra planning
3 author: Atsushi Sakai

```

```

4 link: https://github.com/AtsushiSakai/PythonRobotics/blob/master/PathPlanning/
   Dijkstra/dijkstra.py
5 """
6
7 import matplotlib.pyplot as plt
8 import math
9
10 show_animation = True
11
12
13 class Dijkstra:
14
15     def __init__(self, ox, oy, resolution, robot_radius):
16         """
17         Initialize map for a star planning
18         ox: x position list of Obstacles [m]
19         oy: y position list of Obstacles [m]
20         resolution: grid resolution [m]
21         rr: robot radius[m]
22         """
23
24         self.min_x = None
25         self.min_y = None
26         self.max_x = None
27         self.max_y = None
28         self.x_width = None
29         self.y_width = None
30         self.obstacle_map = None
31
32         self.resolution = resolution
33         self.robot_radius = robot_radius
34         self.calc_obstacle_map(ox, oy)
35         self.motion = self.get_motion_model()
36
37     class Node:
38         def __init__(self, x, y, cost, parent_index):
39             self.x = x # index of grid
40             self.y = y # index of grid
41             self.cost = cost
42             self.parent_index = parent_index # index of previous Node
43
44         def __str__(self):
45             return str(self.x) + "," + str(self.y) + "," + str(
46                 self.cost) + "," + str(self.parent_index)
47
48     def planning(self, sx, sy, gx, gy):
49         """
50         dijkstra path search
51         input:
52             s_x: start x position [m]
53             s_y: start y position [m]
54             gx: goal x position [m]
55             gx: goal x position [m]
56         output:
57             rx: x position list of the final path
58             ry: y position list of the final path
59         """
60
61         start_node = self.Node(self.calc_xy_index(sx, self.min_x),
62                                self.calc_xy_index(sy, self.min_y), 0.0, -1)
63         goal_node = self.Node(self.calc_xy_index(gx, self.min_x),
64                               self.calc_xy_index(gy, self.min_y), 0.0, -1)

```

```

65 open_set, closed_set = dict(), dict()
66 open_set[self.calc_index(start_node)] = start_node
67
68
69 while 1:
70     c_id = min(open_set, key=lambda o: open_set[o].cost)
71     current = open_set[c_id]
72
73     # show graph
74     if show_animation: # pragma: no cover
75         plt.plot(self.calc_position(current.x, self.min_x),
76                 self.calc_position(current.y, self.min_y), "xc")
77         # for stopping simulation with the esc key.
78         plt.gcf().canvas.mpl_connect(
79             'key_release_event',
80             lambda event: [exit(0) if event.key == 'escape' else None])
81         if len(closed_set.keys()) % 10 == 0:
82             plt.pause(0.001)
83
84     if current.x == goal_node.x and current.y == goal_node.y:
85         print("Find goal")
86         goal_node.parent_index = current.parent_index
87         goal_node.cost = current.cost
88         break
89
90     # Remove the item from the open set
91     del open_set[c_id]
92
93     # Add it to the closed set
94     closed_set[c_id] = current
95
96     # expand search grid based on motion model
97     for move_x, move_y, move_cost in self.motion:
98         node = self.Node(current.x + move_x,
99                           current.y + move_y,
100                           current.cost + move_cost, c_id)
101         n_id = self.calc_index(node)
102
103         if n_id in closed_set:
104             continue
105
106         if not self.verify_node(node):
107             continue
108
109         if n_id not in open_set:
110             open_set[n_id] = node # Discover a new node
111         else:
112             if open_set[n_id].cost >= node.cost:
113                 # This path is the best until now. record it!
114                 open_set[n_id] = node
115
116 rx, ry = self.calc_final_path(goal_node, closed_set)
117
118 return rx, ry
119
120 def calc_final_path(self, goal_node, closed_set):
121     # generate final course
122     rx, ry = [self.calc_position(goal_node.x, self.min_x)], [
123         self.calc_position(goal_node.y, self.min_y)]
124     parent_index = goal_node.parent_index
125     while parent_index != -1:
126         n = closed_set[parent_index]

```

```

127         rx.append(self.calc_position(n.x, self.min_x))
128         ry.append(self.calc_position(n.y, self.min_y))
129         parent_index = n.parent_index
130
131         return rx, ry
132
133     def calc_position(self, index, minp):
134         pos = index * self.resolution + minp
135         return pos
136
137     def calc_xy_index(self, position, minp):
138         return round((position - minp) / self.resolution)
139
140     def calc_index(self, node):
141         return (node.y - self.min_y) * self.x_width + (node.x - self.min_x)
142
143     def verify_node(self, node):
144         px = self.calc_position(node.x, self.min_x)
145         py = self.calc_position(node.y, self.min_y)
146
147         if px < self.min_x:
148             return False
149         if py < self.min_y:
150             return False
151         if px >= self.max_x:
152             return False
153         if py >= self.max_y:
154             return False
155
156         if self.obstacle_map[node.x][node.y]:
157             return False
158
159         return True
160
161     def calc_obstacle_map(self, ox, oy):
162
163         self.min_x = round(min(ox))
164         self.min_y = round(min(oy))
165         self.max_x = round(max(ox))
166         self.max_y = round(max(oy))
167         print("min_x:", self.min_x)
168         print("min_y:", self.min_y)
169         print("max_x:", self.max_x)
170         print("max_y:", self.max_y)
171
172         self.x_width = round((self.max_x - self.min_x) / self.resolution)
173         self.y_width = round((self.max_y - self.min_y) / self.resolution)
174         print("x_width:", self.x_width)
175         print("y_width:", self.y_width)
176
177         # obstacle map generation
178         self.obstacle_map = [[False for _ in range(self.y_width)]
179                               for _ in range(self.x_width)]
180         for ix in range(self.x_width):
181             x = self.calc_position(ix, self.min_x)
182             for iy in range(self.y_width):
183                 y = self.calc_position(iy, self.min_y)
184                 for iox, ioy in zip(ox, oy):
185                     d = math.hypot(iox - x, ioy - y)
186                     if d <= self.robot_radius:
187                         self.obstacle_map[ix][iy] = True
188                         break

```

```

189
190 @staticmethod
191 def get_motion_model():
192     # dx, dy, cost
193     motion = [[1, 0, 1],
194               [0, 1, 1],
195               [-1, 0, 1],
196               [0, -1, 1],
197               [-1, -1, math.sqrt(2)],
198               [-1, 1, math.sqrt(2)],
199               [1, -1, math.sqrt(2)],
200               [1, 1, math.sqrt(2)]]
201
202     return motion
203
204
205 def main():
206     print(__file__ + " start!!")
207
208     # start and goal position
209     sx = -5.0 # [m]
210     sy = -5.0 # [m]
211     gx = 50.0 # [m]
212     gy = 50.0 # [m]
213     grid_size = 2.0 # [m]
214     robot_radius = 1.0 # [m]
215
216     # set obstacle positions
217     ox, oy = [], []
218     for i in range(-10, 60):
219         ox.append(i)
220         oy.append(-10.0)
221     for i in range(-10, 60):
222         ox.append(60.0)
223         oy.append(i)
224     for i in range(-10, 61):
225         ox.append(i)
226         oy.append(60.0)
227     for i in range(-10, 61):
228         ox.append(-10.0)
229         oy.append(i)
230     for i in range(-10, 40):
231         ox.append(20.0)
232         oy.append(i)
233     for i in range(0, 40):
234         ox.append(40.0)
235         oy.append(60.0 - i)
236
237     if show_animation: # pragma: no cover
238         plt.plot(ox, oy, ".k")
239         plt.plot(sx, sy, "og")
240         plt.plot(gx, gy, "xb")
241         plt.grid(True)
242         plt.axis("equal")
243
244     dijkstra = Dijkstra(ox, oy, grid_size, robot_radius)
245     rx, ry = dijkstra.planning(sx, sy, gx, gy)
246
247     if show_animation: # pragma: no cover
248         plt.plot(rx, ry, "-r")
249         plt.pause(0.01)
250         plt.show()

```

```

251
252
253 if __name__ == '__main__':
254     main()

```

Listing A.7: Grid based Dijkstra planning (Atsushi's project)(subsection 3.3.3)

```

1 class Dijkstra:
2     """
3     Modified by Weijian Yang, email: weijiany@stud.ntnu.no
4     Function: find the efficient path for a given velocity
5     Method: Dijkstra
6     Parameters:
7         ox: x position list of boundaries (Obstacles)
8         oy: y position list of boundaries (Obstacles)
9         sx: x position of the start point
10        sy: y position of the start point
11        gx: x position of the goal point
12        gy: y position of the goal point
13        CF_N: costs of movements in north direction
14        CF_E: costs of movements in east direction
15        CF_S: costs of movements in south direction
16        CF_W: costs of movements in west direction
17    Return values:
18        rx: x position list of current path
19        ry: y position list of current path
20        goal_node.cost: total cost of current path
21    """
22    def __init__(self, ox, oy):
23        # initialize parameters
24        self.min_x = None
25        self.max_x = None
26        self.min_y = None
27        self.max_y = None
28        self.x_grid_num = None
29        self.y_grid_num = None
30        self.obstacle_map = None
31
32        self.calc_obstacle_grid_map(ox, oy)
33
34    def calc_obstacle_grid_map(self, ox, oy):
35        """ build obstacle map """
36        """
37        Parameters:
38            ox: x position list of boundaries (Obstacles)
39            oy: y position list of boundaries (Obstacles)
40        """
41        # 1. get boundaries' values of the environment
42        self.min_x = round(min(ox))
43        self.max_x = round(max(ox))
44        self.min_y = round(min(oy))
45        self.max_y = round(max(oy))
46
47        # 2. calculate needed numbers of x,y in map
48        self.x_grid_num = round(self.max_x - self.min_x)
49        self.y_grid_num = round(self.max_y - self.min_y)
50
51        # 3. obstacle map generation
52        self.obstacle_map = [[False for _ in range(self.x_grid_num)] for _ in
53                               range(self.y_grid_num)]
54
55    def planning(self, sx, sy, gx, gy, CF_N, CF_E, CF_S, CF_W):
56        """ dijkstra path search """

```



```

56         """
57         Parameters:
58             sx: x position of the start point
59             sy: y position of the start point
60             gx: x position of the goal point
61             gy: y position of the goal point
62             CF_N: costs of movements in north direction
63             CF_E: costs of movements in east direction
64             CF_S: costs of movements in south direction
65             CF_W: costs of movements in west direction
66         Return values:
67             rx: x position list of current path
68             ry: y position list of current path
69             goal_node.cost: total cost of current path
70         """
71         # 1. get start_node, goal_node
72         sx_index = self.calc_xy_index(sx, self.min_x)
73         sy_index = self.calc_xy_index(sy, self.min_y)
74         gx_index = self.calc_xy_index(gx, self.min_x)
75         gy_index = self.calc_xy_index(gy, self.min_y)
76         start_node = self.Node(sx_index, sy_index, 0.0, -1)
77         goal_node = self.Node(gx_index, gy_index, 0.0, -1)
78
79         # 2. initialize open_set, close_set, put start_node into open_set
80         open_set, close_set = dict(), dict()
81         open_set[self.calc_index(start_node)] = start_node
82
83         # 3. search
84         while True:
85             # (1). choose the node whose cost is minimum in open_set
86             c_id = min(open_set, key=lambda o: open_set[o].cost)
87             current = open_set[c_id]
88
89             if show:
90                 plt.plot(self.calc_position(current.x, self.min_x),
91                         self.calc_position(current.y, self.min_y), "xc")
92                 # for stopping simulation with the esc key.
93                 # plt.gcf().canvas.mpl_connect(
94                 #     'key_release_event',
95                 #     lambda event: [exit(0) if event.key == 'escape' else None])
96                 if len(close_set.keys()) % 10 == 0:
97                     plt.pause(0.001)
98
99             # (2). determine whether the current node is the end point
100             if current.x == goal_node.x and current.y == goal_node.y:
101                 goal_node.parent_index = current.parent_index
102                 goal_node.cost = current.cost
103                 break
104
105             # (3). remove the current node from the open set, add it to the
106             closed set
107             del open_set[c_id]
108             close_set[c_id] = current
109
110             # (4). expand search grid based on motion model
111             self.robot_motion = self.get_motion_model(current.x, current.y, CF_N,
112             CF_E, CF_S, CF_W)
113             for move_x, move_y, move_cost in self.robot_motion:
114                 node = self.Node(current.x + move_x,
115                                 current.y + move_y,
116                                 current.cost + move_cost, c_id)
117                 n_id = self.calc_index(node)

```

```

116         if n_id in close_set:
117             continue
118
119         if not self.verify_node(node):
120             continue
121
122         if n_id not in open_set:
123             open_set[n_id] = node    # discover a new node
124         else:
125             if open_set[n_id].cost >= node.cost:
126                 # This path is the best until now. record it!
127                 open_set[n_id] = node
128
129     rx, ry = self.calc_final_path(goal_node, close_set)
130
131     return rx, ry, goal_node.cost
132
133 def calc_final_path(self, goal_node, close_set):
134     """ generate final course """
135     rx = [self.calc_position(goal_node.x, self.min_x)]
136     ry = [self.calc_position(goal_node.y, self.min_y)]
137
138     parent_index = goal_node.parent_index
139     while parent_index != -1:
140         n = close_set[parent_index]
141         rx.append(self.calc_position(n.x, self.min_x))
142         ry.append(self.calc_position(n.y, self.min_y))
143         parent_index = n.parent_index
144
145     return rx, ry
146
147 class Node:
148     def __init__(self, x, y, cost, parent_index):
149         self.x = x
150         self.y = y
151         self.cost = cost        # g(n)
152         self.parent_index = parent_index
153
154     #
155     # def __str__(self):
156     #     return str(self.x) + "," + str(self.y) + "," + str(self.cost) + "," + str(self.parent_index)
157     """
158     These 3 functions calc_index, calc_xy_index and calc_position are used for
159     coordinate system transformation
160     However, in this project, they are useless because values of x and y in grid
161     map coordinates and xy map are the same.
162     """
163     def calc_index(self, node):
164         index = node.y * self.x_grid_num + node.x
165         return index
166
167     def calc_xy_index(self, pos, min_p):
168         index = round(pos - min_p)
169         return index
170
171     def calc_position(self, index, min_p):
172         pos = min_p + index
173         return pos
174
175     def verify_node(self, node):
176         """ check whether the current position is appropriate """

```

```

175     px = self.calc_position(node.x, self.min_x)
176     py = self.calc_position(node.y, self.min_y)
177
178     if px < self.min_x or px > self.max_x:
179         return False
180     if py < self.min_y or py > self.max_y:
181         return False
182
183     return True
184
185     @staticmethod
186     def get_motion_model(x,y,CF_N,CF_E,CF_S,CF_W):
187         # dx, dy, cost
188         data_x=x
189         data_y=y
190         model = [
191             [0, 1, CF_N[data_x,data_y+1]],          # North
192             [0, -1,CF_S[data_x,data_y-1]],          # South
193             [-1, 0,CF_E[data_x-1,data_y]],          # East
194             [1, 0, CF_W[data_x+1,data_y]],          # West
195         ]
196         return model

```

Listing A.8: Function for finding the efficient path under a given speed (subsection 3.3.3)

```

1  ...
2  """ Build boundary """
3  ox, oy = [], []
4  for i in range(0, xcol):          #north boundary
5      ox.append(i)
6      oy.append(xcol)
7  for i in range(0, yrow):          #east boundary
8      ox.append(0)
9      oy.append(i)
10 for i in range(0, xcol):           #south boundary
11     ox.append(i)
12     oy.append(0)
13 for i in range(0, yrow):           #west boundary
14     ox.append(yrow)
15     oy.append(i)
16 dijkstra = Dijkstra(ox, oy)
17
18 T_cost=np.mat(np.zeros((1,num_iter)))
19 TotalCost=T_cost
20 show = False
21 for i in range(num_iter):
22     length=(abs(sx-gx)+abs(sy-gy))+i
23     """
24     find minimum cost in each velocity
25     Vs_min=(abs(sx-gx)+abs(sy-gy))/t
26     Vs_max=xcol*yrow/t
27     """
28     Vs=float(length*dist/t)
29     delta_T=dist/Vs
30
31     """calculate cost for each given velocity"""
32     CF_Wind_N,CF_Wind_E,CF_Wind_S,CF_Wind_W = WindResCosFun(Vs,Axv,rho_air,wind_u
33         ,wind_v,Cair_extend,delta_T)
34     CF_m=TotalResCosFun(Vs,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,
35         rho_sea,nu_sea,g,delta_T)
36     CF_Wave_N,CF_Wave_E,CF_Wave_S,CF_Wave_W=WaveResCosFun(Vs,Lbwl,B,rho_sea,

```

```

wave_d,wave_h,delta_T)
35 CF_N=CF_Wind_N+CF_Wave_N+CF_m
36 CF_E=CF_Wind_E+CF_Wave_E+CF_m
37 CF_S=CF_Wind_S+CF_Wave_S+CF_m
38 CF_W=CF_Wind_W+CF_Wave_W+CF_m
39
40 """find an efficient path for each given velocity"""
41 rx, ry, T_cost[0,i] = dijkstra.planning(sx, sy, gx, gy,CF_N,CF_E,CF_S,CF_W)
42
43 """check task time"""
44 if len(rx)*delta_T>t:
45     # print('The ship cannot reach the goal point in given time with this
46     speed',Vs , ' [m/s]')
47     print('The ship cannot reach the goal point in given time with this speed
48     ',Vs/0.5144 , ' [knot]')
49     TotalCost[0,i]=0
50 else:
51     # print('The ship could reach the goal point in given time with this
52     speed',Vs , ' [m/s]')
53     print('The ship could reach the goal point in given time with this speed'
54     ',Vs/0.5144 , ' [knot]')
55     TotalCost[0,i]=T_cost[0,i]
56 if np.max(TotalCost)==0:
57     print('The effective path could not be found after ',num_iter,' iterations' )
58 else:
59
60 """compare total costs of paths"""
61 minx,miny= np.where(TotalCost == np.min(TotalCost[np.nonzero(TotalCost)]))
62
63 """get optimum velocity"""
64 length_E=(abs(sx-gx)+abs(sy-gy))+miny
65 Vs_E=float(length_E*dist/t)
66 delta_TE=dist/Vs_E
67 CFE_Wind_N,CFE_Wind_E,CFE_Wind_S,CFE_Wind_W = WindResCosFun(Vs_E,Axv,rho_air,
68 wind_u,wind_v,Cair_extend,delta_T)
69 CFE_m=TotalResCosFun(Vs_E,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,
70 rho_sea,nu_sea,g,delta_T)
71 CFE_Wave_N,CFE_Wave_E,CFE_Wave_S,CFE_Wave_W=WaveResCosFun(Vs_E,Lbwl,B,rho_sea
72 ,wave_d,wave_h,delta_T)
73 CF_N=CFE_Wind_N+CFE_Wave_N+CFE_m
74 CF_E=CFE_Wind_E+CFE_Wave_E+CFE_m
75 CF_S=CFE_Wind_S+CFE_Wave_S+CFE_m
76 CF_W=CFE_Wind_W+CFE_Wave_W+CFE_m
77 show = True
78 if show:
79     plt.plot(ox, oy, '.k')
80     plt.plot(sx, sy, 'og')
81     plt.plot(gx, gy, 'or')
82     # plt.grid('True')
83     plt.axis('equal')
84     # plt.show()
85
86 """get optimum path for this optimum velocity"""
87 rx, ry, TotalCost_min = dijkstra.planning(sx, sy, gx, gy,CF_N,CF_E,CF_S,
88 CF_W)
89 if show:
90     print('Find the effective path' )
91 ...

```

Listing A.9: Codes for choosing the optimum speed (subsection 3.3.4)

A.2 Testing codes

```
1 """
2 Program tests
3 Course: TMR 4345
4 Project: Ship Routing
5 Author: Weijian Yang, email: weijiany@stud.ntnu.no
6 """
7 import math
8 import numpy as np
9 import pandas as pd
10 import matplotlib.pyplot as plt
11 ## Function for importing data from Csv file
12 ## numpy -> delimiter = ";"
13 def ReadCsvWind(filepath,xcol,yrow):
14     file = open(filepath, "rb")
15     filedata=np.loadtxt(file, delimiter=";")
16     file.close()
17     filearray=np.array(filedata)
18     # +2: just in case for that: data is not enough for grid
19     num_row=xcol+2
20     num_col=yrow+2
21     dataname = np.zeros(shape=(num_row,num_col))
22     for i in range(num_row):
23         for j in range(num_col):
24             dataname[i,j]=filearray[i, j]
25     return dataname
26 ## pandas -> delimiter = "rows & columns"
27 def ReadCsvWave(filepath,xcol,yrow):
28     file =pd.read_csv(filepath)
29     # +2: just in case for that: data is not enough for grid
30     num_row=xcol+2
31     num_col=yrow+2
32     dataname = file.values[0:num_row,0:num_col]
33     return dataname
34 ## Function for calculating wind resistance:
35 ##  $R=0.5 \cdot \rho_{\text{air}} \cdot A_{\text{vx}} \cdot [\text{Cair}(\text{relative wind angle}) \cdot V(\text{relative wind velocity})^2 - \text{Cair}(0) \cdot V(\text{ship velocity})^2]$ 
36 def WindResCosFun(Vs,Axv,rho_air,wind_u,wind_v,Cair_extend,dt):
37     angle_ship=np.mat([0,90,180,270]) #Angle of ship: North, East, South and West
38     for k in range(angle_ship.shape[1]):
39         vel_wind_squire=np.mat(np.zeros(wind_v.shape))
40         vel_wt_squire=np.mat(np.zeros(wind_u.shape))
41         angle_wt=np.mat(np.zeros(wind_u.shape))
42         angle_wt_rel=np.mat(np.zeros(wind_v.shape))
43         angle_rel=np.mat(np.zeros(wind_u.shape))
44         Cx=np.mat(np.zeros(wind_v.shape))
45         if k == 0:
46             wind_u_north=wind_u+Vs
47             for i in range(wind_u.shape[0]):
48                 for j in range(wind_v.shape[1]):
49                     vel_wt_squire[i,j]=wind_u_north[i, j]**2+wind_v[i, j]**2 #
50             Vwt:true wind velocity
51             if vel_wt_squire[i,j]==0:
52                 angle_wt[i,j]=0
53             else:
54                 angle_wt[i,j]=math.degrees(math.acos(wind_v[i,j]/math.sqrt(vel_wt_squire[i,j]))) #Awt:true wind direction
55             angle_wt_rel[i,j]=angle_wt[i,j]-angle_ship[0,k]+180 #
56     Awt-Aship
```

```

56         vel_wind_squire[i,j]=vel_wt_squire[i,j]+Vs**2+math.sqrt(
vel_wt_squire[i,j])*Vs*math.cos(math.radians(angle_wt_rel[i,j])) #V_WRef
57         nume=math.sqrt(vel_wt_squire[i,j])*math.sin(math.radians(
angle_wt_rel[i,j]))
58         deno=Vs+math.sqrt(vel_wt_squire[i,j])*math.cos(math.radians(
angle_wt_rel[i,j]))
59         if deno==0:
60             if nume<0:
61                 angle_rel[i,j]=-90
62             if nume>0:
63                 angle_rel[i,j]=90
64             if nume==0:
65                 angle_rel[i,j]=0
66         elif deno<0:
67             angle_rel[i,j]=math.degrees(math.atan(nume/deno))+180
#A_WRef
68         else:
69             angle_rel[i,j]=math.degrees(math.atan(nume/deno))
70         if angle_rel[i,j]< 0:
71             angle_rel[i,j]=angle_rel[i,j]+360
72
73         weight=angle_rel[i,j]/10
74         index=int(angle_rel[i,j]/10)
75         Cx[i,j]=(weight-index)*Cair_extend[0,index+1]+(1-weight+index
)*Cair_extend[0,index]
76         Rwind_N=0.5*rho_air*Axv*(Cx*vel_wind_squire-Cair_extend[0,0]*Vs**2)
77         CF_Wind_North=Rwind_N*Vs*dt
78
79         if k == 1:
80             wind_v_east=wind_v+Vs
81             for i in range(wind_u.shape[0]):
82                 for j in range(wind_v.shape[1]):
83                     vel_wt_squire[i,j]=wind_u[i, j]**2+wind_v_east[i, j]**2 #
Vwt:true wind velocity
84                     if vel_wt_squire[i,j]==0:
85                         angle_wt[i,j]=0
86                     else:
87                         angle_wt[i,j]=math.degrees(math.asin(wind_v_east[i,j]/
math.sqrt(vel_wt_squire[i,j]))) #Awt:true wind direction
88
89                     angle_wt_rel[i,j]=angle_wt[i,j]-angle_ship[0,k]+180 #
Awt-Aship
90                     vel_wind_squire[i,j]=vel_wt_squire[i,j]+Vs**2+math.sqrt(
vel_wt_squire[i,j])*Vs*math.cos(math.radians(angle_wt_rel[i,j])) #V_WRef
91                     nume=math.sqrt(vel_wt_squire[i,j])*math.sin(math.radians(
angle_wt_rel[i,j]))
92                     deno=Vs+math.sqrt(vel_wt_squire[i,j])*math.cos(math.radians(
angle_wt_rel[i,j]))
93                     if deno==0:
94                         if nume<0:
95                             angle_rel[i,j]=-90
96                         if nume>0:
97                             angle_rel[i,j]=90
98                         if nume==0:
99                             angle_rel[i,j]=0
100                     elif deno<0:
101                         angle_rel[i,j]=math.degrees(math.atan(nume/deno))+180
#A_WRef
102                     else:
103                         angle_rel[i,j]=math.degrees(math.atan(nume/deno))
104                     if angle_rel[i,j]< 0:
105                         angle_rel[i,j]=angle_rel[i,j]+360

```

```

106         weight=angle_rel[i,j]/10
107         index=int(angle_rel[i,j]//10)
108         Cx[i,j]=(weight-index)*Cair_extend[0,index+1]+(1-weight+index
109 )*Cair_extend[0,index]
110         Rwind_E=0.5*rho_air*Axv*(Cx*vel_wind_squre-Cair_extend[0,0]*Vs**2)
111         CF_Wind_East=Rwind_E*Vs*dt
112
113     if k == 2:
114         wind_u_south=wind_u-Vs
115         for i in range(wind_u.shape[0]):
116             for j in range(wind_v.shape[1]):
117                 vel_wt_squre[i,j]=wind_u_south[i,j]**2+wind_v[i,j]**2    #
118 Vwt:true wind velocity
119                 if vel_wt_squre[i,j]==0:
120                     angle_wt[i,j]=0
121                 else:
122                     angle_wt[i,j]=math.degrees(math.acos(wind_v[i,j]/math.
123 sqrt(vel_wt_squre[i,j]))) #Awt:true wind direction
124
125                 angle_wt_rel[i,j]=angle_wt[i,j]-angle_ship[0,k]+180    #
126 Awt-Aship
127                 vel_wind_squre[i,j]=vel_wt_squre[i,j]+Vs**2+math.sqrt(
128 vel_wt_squre[i,j])*Vs*math.cos(math.radians(angle_wt_rel[i,j])) #V_WRef
129                 nume=math.sqrt(vel_wt_squre[i,j])*math.sin(math.radians(
130 angle_wt_rel[i,j]))
131                 deno=Vs+math.sqrt(vel_wt_squre[i,j])*math.cos(math.radians(
132 angle_wt_rel[i,j]))
133                 if deno==0:
134                     if nume<0:
135                         angle_rel[i,j]=-90
136                     if nume>0:
137                         angle_rel[i,j]=90
138                     if nume==0:
139                         angle_rel[i,j]=0
140                 elif deno<0:
141                     angle_rel[i,j]=math.degrees(math.atan(nume/deno))+180
142 #A_WRef
143                 else:
144                     angle_rel[i,j]=math.degrees(math.atan(nume/deno))
145                 if angle_rel[i,j]< 0:
146                     angle_rel[i,j]=angle_rel[i,j]+360
147
148         weight=angle_rel[i,j]/10
149         index=int(angle_rel[i,j]//10)
150         Cx[i,j]=(weight-index)*Cair_extend[0,index+1]+(1-weight+index
151 )*Cair_extend[0,index]
152         Rwind_S=0.5*rho_air*Axv*(Cx*vel_wind_squre-Cair_extend[0,0]*Vs**2)
153         CF_Wind_South=Rwind_S*Vs*dt
154
155     if k == 3:
156         wind_v_west=wind_v-Vs
157         for i in range(wind_u.shape[0]):
158             for j in range(wind_v.shape[1]):
159                 vel_wt_squre[i,j]=wind_u[i, j]**2+wind_v_west[i, j]**2    #
160 Vwt:true wind velocity
161                 if vel_wt_squre[i,j]==0:
162                     angle_wt[i,j]=0
163                 else:
164                     angle_wt[i,j]=math.degrees(math.asin(wind_v_west[i,j]/
165 math.sqrt(vel_wt_squre[i,j]))) #Awt:true wind direction

```

```

157         angle_wt_rel[i,j]=angle_wt[i,j]-angle_ship[0,k]+180 #
158     Awt-Aship
159         vel_wind_squire[i,j]=vel_wt_squire[i,j]+Vs**2+math.sqrt(
160         vel_wt_squire[i,j])*Vs*math.cos(math.radians(angle_wt_rel[i,j])) #V_WRef
161         nume=math.sqrt(vel_wt_squire[i,j])*math.sin(math.radians(
162         angle_wt_rel[i,j]))
163         deno=Vs+math.sqrt(vel_wt_squire[i,j])*math.cos(math.radians(
164         angle_wt_rel[i,j]))
165         if deno==0:
166             if nume<0:
167                 angle_rel[i,j]=-90
168             if nume>0:
169                 angle_rel[i,j]=90
170             if nume==0:
171                 angle_rel[i,j]=0
172         elif deno<0:
173             angle_rel[i,j]=math.degrees(math.atan(nume/deno))+180
174         #A_WRef
175         else:
176             angle_rel[i,j]=math.degrees(math.atan(nume/deno))
177             if angle_rel[i,j]< 0:
178                 angle_rel[i,j]=angle_rel[i,j]+360
179
180         weight=angle_rel[i,j]/10
181         index=int(angle_rel[i,j]/10)
182         Cx[i,j]=(weight-index)*Cair_extend[0,index+1]+(1-weight+index
183         )*Cair_extend[0,index]
184         Rwind_W=0.5*rho_air*Axv*(Cx*vel_wind_squire-Cair_extend[0,0]*Vs**2)
185         CF_Wind_West=Rwind_W*Vs*dt
186
187     return CF_Wind_North,CF_Wind_East,CF_Wind_South,CF_Wind_West
188 ## Function for calculating total resistance (only constant velocity)
189 ## Hollenbach Method (only return mean resistance)
190 def TotalResCosFun(Vs,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,rho_sea,
191     nu_sea,g,dt):
192     rho = 1025 #Density of sea water [kg/m^3]
193     gravk = 9.81 #Gravitational constant [m/s^2]
194     nu = 1.1395E-6 #Viscosity of sea water [m/s^2]
195
196     T = (TF+TA)/2
197     """ Calculation of 'Froude length', Lfn """
198     if Los/L < 1:
199         Lfn = Los
200     elif (Los/L >= 1) & (Los/L < 1.1):
201         Lfn = L+2/3*(Los-L)
202     elif Los/L >= 1.1:
203         Lfn = 1.0667*L
204
205     # 'Mean' resistance coefficients
206     a = np.mat([-0.3382, 0.8086, -6.0258, -3.5632, 9.4405, 0.0146, 0, 0, 0, 0])
207     #a1 means a[0,0]
208     b = np.mat([-0.57424, 13.3893, 90.5960],[4.6614, -39.721,
209     -351.483],[-1.14215, -12.3296, 459.254])) #b12 means b[0,1]
210     d = np.mat([0.854, -1.228, 0.497])
211     e = np.mat([2.1701, -0.1602])
212     f = np.mat([0.17, 0.20, 0.60])
213     g = np.mat([0.642, -0.635, 0.150])
214
215     Fn = Vs/((gravk*Lfn)**0.5) #Froude's number
216     Fnkrit_help0=np.mat([1,CB,CB**2]) # Build Matrix for using transpose
217     : Fnkrit_help0.T
218     Fnkrit_help1 = d*Fnkrit_help0.T # Fnkrit_help1=[[x]] Matrix

```



```

209     type
    Fnkrit=Fnkrit_help1[0,0]                                # Fnkrit=x
        Float type
210     c1 = Fn/Fnkrit
211     Rns = Vs*L/nu                                          #Reynold's number for ship
212     if Rns == 0:                                          #Rns=0,log
        would get stuck
213         CFs =0
214     else :
215         CFs = 0.075/(math.log10(Rns)-2)**2              #ITTC friction line for ship
216
217
218     """ Calculation of C_R for given ship"""
219     # Mean value
220     CRFnkrit = max(1.0,(Fn/Fnkrit)**c1)
221     kL = e[0,0]*L**(e[0,1])
222
223     # There is an error in the hollenbach paper and in Minsaas' 2003 textbook,
224     # which is corrected in this formula by dividing by 10
225     CRstandard_help0=np.mat([1,Fn,Fn**2])
226     CRstandard_help1=Fnkrit_help0*(b*CRstandard_help0.T)/10
227     CRstandard=CRstandard_help1[0,0]
228
229     #prod([T/B B/L Los/Lwl Lwl/L (1+(TA-TF)/L) Dp/TA (1+NRud) (1+NBrac) (1+NBoss)
230     # (1+NThr)].^a)
231     prod_help=np.mat([T/B,B/L,Los/Lwl,Lwl/L,1+(TA-TF)/L,Dp/TA,1+NRud,1+NBrac,1+
232     NBoss,1+NThr])
233     prod_help1=np.mat(np.ones((1,10)))                    #build a Matrix
234     [1,10]
235     for j in range(a.size):                                #prod_help
236         =[prod_help[0,i].^a_min[0,i]]
237         prod_help1[0,j]=prod_help[0,j]**a[0,j]
238     prod_help2=np.prod(prod_help1,axis = 1)                #prod function
239     prod=prod_help2[0,0]
240
241     CR_hollenbach = CRstandard*CRFnkrit*kL*prod
242     CR = CR_hollenbach*B*T/S                                #Resistance coefficient, scaled for
243     wetted surface
244     C_Ts = CFs + CR                                          #Total resistance coeff.
245     ship
246     R_T_mean = C_Ts*rho/2*Vs**2*S                          #Total resistance to the ship
247
248     Fn_min=min(f[0,0],f[0,0]+f[0,1]*(f[0,2]-CB))
249     Fn_max=g[0,0]+g[0,1]*CB+g[0,2]*CB**3
250
251     if Fn>Fn_max:
252         #R_T=h1*R_T_mean
253         R_T=1.204*R_T_mean #h1=1.204
254     elif Fn<Fn_min:
255         #CRFnkrit=kL=1.0
256         CR_min=CRstandard*prod*B*T/S
257         R_T= (CFs + CR_min)*rho/2*Vs**2*S
258     else:
259         R_T=R_T_mean
260
261     C_T = R_T*Vs*dt
262     return C_T
263
264     ## Function for calculating added resistance (-45 degrees ~ 45 degrees)
265     ## R=1/16*rho_sea*g*H^2*B*sqrt(B/Lbwl)
266 def WaveResCosFun(Vs,Lbwl,B,rho_sea,wave_d,wave_h,dt):
267     angle_ship=np.mat([0,90,180,270]) #Angle of ship: North, East, South and
268     West

```

```

260     for k in range(angle_ship.shape[1]):
261         angle_rel=np.mat(np.zeros(wave_d.shape))
262         R_wave=np.mat(np.zeros(wave_h.shape))
263         for i in range(wave_d.shape[0]):
264             for j in range(wave_h.shape[1]):
265                 angle_rel[i,j]=wave_d[i,j]-angle_ship[0,k]
266                 while angle_rel[i,j]<-180 or angle_rel[i,j]>=180:
267                     if angle_rel[i,j]< -180:
268                         angle_rel[i,j]=angle_rel[i,j]+360
269                     if angle_rel[i,j]>=180:
270                         angle_rel[i,j]=angle_rel[i,j]-360
271
272                 if angle_rel[i,j]<-45 or angle_rel[i,j]>45:
273                     R_wave[i,j]=0
274                 else:
275                     R_wave[i,j]=rho_sea*g*B*math.sqrt(B/Lbwl)/16*wave_h[i,j]**2
276                 C_wave=R_wave*Vs*dt
277             if k==0:
278                 CF_Wave_N=C_wave
279             if k==1:
280                 CF_Wave_E=C_wave
281             if k==2:
282                 CF_Wave_S=C_wave
283             if k==0:
284                 CF_Wave_W=C_wave
285         return CF_Wave_N,CF_Wave_E,CF_Wave_S,CF_Wave_W
286 ## Dijkstra's alogrithm (find path)
287 class Dijkstra:
288     def __init__(self, ox, oy):
289         # initialize parameters
290         self.min_x = None
291         self.max_x = None
292         self.min_y = None
293         self.max_y = None
294         self.x_grid_num = None
295         self.y_grid_num = None
296         self.obstacle_map = None
297         self.calc_obstacle_grid_map(ox, oy)
298
299     def calc_obstacle_grid_map(self, ox, oy):
300         """ build obstacle map """
301         # 1. get boundaries' values of the environment
302         self.min_x = round(min(ox))
303         self.max_x = round(max(ox))
304         self.min_y = round(min(oy))
305         self.max_y = round(max(oy))
306
307         # 2. calculate needed numbers of x,y in map
308         self.x_grid_num = round(self.max_x - self.min_x)
309         self.y_grid_num = round(self.max_y - self.min_y)
310
311         # 3. obstacle map generation
312         self.obstacle_map = [[False for _ in range(self.x_grid_num)] for _ in
range(self.y_grid_num)]
313
314     def planning(self, sx, sy, gx, gy,CF_N,CF_E,CF_S,CF_W):
315         """ dijkstra path search """
316
317         # 1. get start_node, goal_node
318         sx_index = self.calc_xy_index(sx, self.min_x)
319         sy_index = self.calc_xy_index(sy, self.min_y)
320         gx_index = self.calc_xy_index(gx, self.min_x)

```

```

321     gy_index = self.calc_xy_index(gy, self.min_y)
322     start_node = self.Node(sx_index, sy_index, 0.0, -1)
323     goal_node = self.Node(gx_index, gy_index, 0.0, -1)
324
325     # 2. initialize open_set, close_set, put start_node into open_set
326     open_set, close_set = dict(), dict()
327     open_set[self.calc_index(start_node)] = start_node
328
329     # 3. search
330     while True:
331         # (1). choose the node whose cost is minimum in open_set
332         c_id = min(open_set, key=lambda o: open_set[o].cost)
333         current = open_set[c_id]
334
335         if show:
336             plt.plot(self.calc_position(current.x, self.min_x),
337                     self.calc_position(current.y, self.min_y), "xc")
338             # for stopping simulation with the esc key.
339             # plt.gcf().canvas.mpl_connect(
340             #     'key_release_event',
341             #     lambda event: [exit(0) if event.key == 'escape' else None])
342             if len(close_set.keys()) % 10 == 0:
343                 plt.pause(0.001)
344
345         # (2). determine whether the current node is the end point
346         if current.x == goal_node.x and current.y == goal_node.y:
347             goal_node.parent_index = current.parent_index
348             goal_node.cost = current.cost
349             break
350
351         # (3). remove the current node from the open set, add it to the
352         closed set
353         del open_set[c_id]
354         close_set[c_id] = current
355
356         # (4). expand search grid based on motion model
357         self.robot_motion = self.get_motion_model(current.x, current.y, CF_N,
358             CF_E, CF_S, CF_W)
359         for move_x, move_y, move_cost in self.robot_motion:
360             node = self.Node(current.x + move_x,
361                             current.y + move_y,
362                             current.cost + move_cost, c_id)
363             n_id = self.calc_index(node)
364
365             if n_id in close_set:
366                 continue
367
368             if not self.verify_node(node):
369                 continue
370
371             if n_id not in open_set:
372                 open_set[n_id] = node # discover a new node
373             else:
374                 if open_set[n_id].cost >= node.cost:
375                     # This path is the best until now. record it!
376                     open_set[n_id] = node
377
378         rx, ry = self.calc_final_path(goal_node, close_set)
379
380     return rx, ry, goal_node.cost
381
382 def calc_final_path(self, goal_node, close_set):

```

```

381     """ generate final course """
382     rx = [self.calc_position(goal_node.x, self.min_x)]
383     ry = [self.calc_position(goal_node.y, self.min_y)]
384
385     parent_index = goal_node.parent_index
386     while parent_index != -1:
387         n = close_set[parent_index]
388         rx.append(self.calc_position(n.x, self.min_x))
389         ry.append(self.calc_position(n.y, self.min_y))
390         parent_index = n.parent_index
391
392     return rx, ry
393
394 class Node:
395     def __init__(self, x, y, cost, parent_index):
396         self.x = x
397         self.y = y
398         self.cost = cost          # g(n)
399         self.parent_index = parent_index
400
401     # def __str__(self):
402     #     return str(self.x) + "," + str(self.y) + "," + str(self.cost) + "," + str(self.parent_index)
403     """
404     These 3 functions calc_index, calc_xy_index and calc_position are used for
405     coordinate system transformation
406     However, in this project, they are useless because values of x and y in grid
407     map coordinates and xy map are the same.
408     """
409     def calc_index(self, node):
410         index = node.y * self.x_grid_num + node.x
411         return index
412
413     def calc_xy_index(self, pos, min_p):
414         index = round(pos - min_p)
415         return index
416
417     def calc_position(self, index, min_p):
418         pos = min_p + index
419         return pos
420
421     def verify_node(self, node):
422         """ check whether the current position is appropriate """
423         px = self.calc_position(node.x, self.min_x)
424         py = self.calc_position(node.y, self.min_y)
425
426         if px < self.min_x or px > self.max_x:
427             return False
428         if py < self.min_y or py > self.max_y:
429             return False
430
431         return True
432
433     @staticmethod
434     def get_motion_model(x, y, CF_N, CF_E, CF_S, CF_W):
435         # dx, dy, cost
436         data_x = x
437         data_y = y
438         model = [
439             [0, 1, CF_N[data_x, data_y+1]],      # North
440             [0, -1, CF_S[data_x, data_y-1]],     # South
441             [-1, 0, CF_E[data_x-1, data_y]],     # East

```

```

440         [1, 0, CF_W[data_x+1,data_y]],          # West
441     ]
442     return model
443 # ## for tasks
444 # ## t,sx,sy,gx,gy,xcol,yrow,num_iter,dist
445 """
446 test 1: time=0.1,num_iter=30          succeed
447 test 2: time=1,num_iter=30            succeed
448 test 3: time=10, num_iter=30          succeed
449 test 4: time=100, num_iter=30         fail
450 test 5: time=100,num_iter=50          succeed
451 """
452 time=100                                #Given Time [h]
453 sx, sy = 3, 3                           #x,y of Start Point
454 gx, gy = 47, 47                         #x,y of Goal Point
455 xcol=50                                 #Number of Columns
456 yrow=50                                 #Number of Rows
457 num_iter=50                             #Limitation of iterations
458 # time=float(input('given time(h):'))
459 t=time*3600                             #Given time[s]
460 dist=1000                              #Distance between neighbour nodes[m]
461 # sx=int(input('start point[x](km):'))
462 # sy=int(input('start point[y](km):'))
463 # gx=int(input('goal point[x](km):'))
464 # gy=int(input('goal point[y](km):'))
465 # # it is a better choice to choose xcol=yrow
466 # xcol=int(input('number of columns(>abs(sy-gy)):'))
467 # yrow=int(input('number of rows(>abs(sx-gx)):'))
468 # num_iter=float(input('Limitation of iterations(max=xcol*yrow):'))
469 # dist=float(input('Distance between neighbour nodes(m):'))
470 # ## for wind resistance
471 # ## Vs,rho_air,Cair,filepath1,filepath2
472 Axv=1600
473 rho_air=1.293
474 filepath1 = "E:/User/Desktop/datalab/u-wind1.csv"
475 filepath2 = "E:/User/Desktop/datalab/v-wind1.csv"
476 # Axv=float(input('Area of maximum transverse section exposed to the wind(m^2):'))
477 # rho_air=float(input('Density of Air (kg/m^3):'))
478 # Cair=np.mat(float(input('wind resistance coefficient(Be careful with
479 # Cair_extend):')))
479 # filepath1=input('filepath of wind_u:')
480 # filepath2=input('filepath of wind_v:')
481 # ## for total resistance
482 # ## Vs,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,rho_sea,nu_sea,g
483 L=200
484 Lwl=195
485 Los=196.5
486 B=33
487 TF=11.6
488 TA=11.4
489 CB=0.855
490 S=10500
491 Dp=7
492 NRud=1
493 NBrac=1
494 NBoss=1
495 NThr=1
496
497 rho_sea = 1025
498 nu_sea = 1.1395E-6
499 g = 9.81

```

```

500 # L=float(input('Length of Ship(m):'))
501 # Lwl=float(input('Length of Water Line(m):'))
502 # Los=float(input('Length over Surface(m):'))
503 # B=float(input('Beam(m):'))
504 # TF=float(input('Draft of Fore Propeller(m):'))
505 # TA=float(input('Draft of Aft Propeller(m):'))
506 # CB=float(input('Block coefficient:'))
507 # S=float(input('Wetted Surface(m^2):'))
508 # Dp=float(input('Propeller diameter(m):'))
509 # NRud=float(input('Number of rudders:'))
510 # NBrac=float(input('Number of brackets:'))
511 # NBoss=float(input('Number of bossings:'))
512 # NThr=float(input('Number of side thrusters:'))
513 # rho_sea=float(input('Density of Sea Water (kg/m^3):'))
514 # nu_sea=float(input('Viscosity of Sea water (m^2/s):'))
515 # g=float(input('Gravitational Constant (m/s^2):'))
516 # ## for added resistance
517 # ## Vs,Lbwl,B,rho_sea,g,filepath3,filepath4
518 Lbwl=38
519 filepath3 = "E:/User/Desktop/datalab/wave_mwd.csv"
520 filepath4 = "E:/User/Desktop/datalab/wave_swh.csv"
521 # Lbwl=float(input('Length of the Bow on the Water Line to 95% of maximum Beam (m
):'))"
522 # filepath3=input('filepath of wave_d:')
523 # filepath4=input('filepath of wave_h:')
524
525 ## Wind Resistance
526 wind_u = ReadCsvWind(filepath1,xcol,yrow)
527 wind_v = ReadCsvWind(filepath2,xcol,yrow)
528
529 """
530 If a new Cair is used, be careful with below functions for Cair_extend
531 Initial data about General Cargo form ITTC (Value range: 0-180 degrees):
532 Cair=[-0.60, -0.87, -1.00, -1.00, -0.88, -0.85, -0.65, -0.42, -0.27, -0.09, 0.09,
0.49, 0.84, 1.39, 1.47, 1.34, 0.92, 0.82]
533 Step 1: Wind resistance coefficient: Data about General Cargo form ITTC, but no
data for 130 degree angle
534 Cair=np.mat([-0.60, -0.87, -1.00, -1.00, -0.88, -0.85, -0.65, -0.42, -0.27,
-0.09, 0.09, 0.49, 0.84, Cair_130,1.39, 1.47, 1.34, 0.92, 0.82])
535 Step 2: Extend this matrix (Value range: 0-360 degrees):
536 """
537 # Step 1
538 Cair_130=0.5*0.84+0.5*1.39 #calculate the
coefficient of 130 degree angle with that of 120 and 140 degree angle
539 Cair=np.mat([-0.60, -0.87, -1.00, -1.00, -0.88, -0.85, -0.65, -0.42, -0.27,
-0.09, 0.09, 0.49, 0.84, Cair_130,1.39, 1.47, 1.34, 0.92, 0.82])
540 # Step 2
541 Cair_extend=np.mat(np.zeros((Cair.size*2-1))) #Value Range of Initial Data
:0-180 degrees
542 Cair_extend[0,Cair.size-1]=Cair[0,Cair.size-1] #Value Range of Extended
Data:0-360 degrees
543 for i in range(Cair.size-1):
544     Cair_extend[0,i]=-Cair[0,i]
545     Cair_extend[0,2*Cair.size-i-2]=-Cair[0,i]
546
547 ## Viscous/Friction+ Wave Resistance
548
549 ## Added Resistance
550 wave_d = ReadCsvWave(filepath3,xcol,yrow)
551 wave_h = ReadCsvWave(filepath4,xcol,yrow)
552
553 """ Build boundary """

```

```

554 ox, oy = [], []
555 for i in range(0, xcol):           #north boundary
556     ox.append(i)
557     oy.append(xcol)
558 for i in range(0, yrow):           #east boundary
559     ox.append(0)
560     oy.append(i)
561 for i in range(0, xcol):           #south boundary
562     ox.append(i)
563     oy.append(0)
564 for i in range(0, yrow):           #west boundary
565     ox.append(yrow)
566     oy.append(i)
567 dijkstra = Dijkstra(ox, oy)
568
569 T_cost=np.mat(np.zeros((1,num_iter)))
570 TotalCost=T_cost
571 show = False
572 for i in range(num_iter):
573     length=(abs(sx-gx)+abs(sy-gy))+i
574     """
575     find minimum cost in each velocity
576     Vs_min=(abs(sx-gx)+abs(sy-gy))/t
577     Vs_max=xcol*yrow/t
578     """
579     Vs=float(length*dist/t)
580     delta_T=dist/Vs
581
582     """calculate cost for each given velocity"""
583     CF_Wind_N,CF_Wind_E,CF_Wind_S,CF_Wind_W = WindResCosFun(Vs,Axv,rho_air,wind_u
,wind_v,Cair_extend,delta_T)
584     CF_m=TotalResCosFun(Vs,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,
rho_sea,nu_sea,g,delta_T)
585     CF_Wave_N,CF_Wave_E,CF_Wave_S,CF_Wave_W=WaveResCosFun(Vs,Lbwl,B,rho_sea,
wave_d,wave_h,delta_T)
586     CF_N=CF_Wind_N+CF_Wave_N+CF_m
587     CF_E=CF_Wind_E+CF_Wave_E+CF_m
588     CF_S=CF_Wind_S+CF_Wave_S+CF_m
589     CF_W=CF_Wind_W+CF_Wave_W+CF_m
590
591     """find an efficient path for each given velocity"""
592     rx, ry, T_cost[0,i] = dijkstra.planning(sx, sy, gx, gy,CF_N,CF_E,CF_S,CF_W)
593
594     """check task time"""
595     if len(rx)*delta_T>t:
596         # print('The ship cannot reach the goal point in given time with this
speed',Vs , ' [m/s]')
597         print('The ship cannot reach the goal point in given time with this speed
',Vs/0.5144 , ' [knot]')
598         TotalCost[0,i]=0
599     else:
600         # print('The ship could reach the goal point in given time with this
speed',Vs , ' [m/s]')
601         print('The ship could reach the goal point in given time with this speed'
,Vs/0.5144 , ' [knot]')
602         TotalCost[0,i]=T_cost[0,i]
603 if np.max(TotalCost)==0:
604     print('The effective path could not be found after ',num_iter,' iterations' )
605 else:
606
607     """compare total costs of paths"""
608     minx,miny= np.where(TotalCost == np.min(TotalCost[np.nonzero(TotalCost)]))

```

```

609
610 """get optimum velocity"""
611 length_E=(abs(sx-gx)+abs(sy-gy))+miny
612 Vs_E=float(length_E*dist/t)
613 delta_TE=dist/Vs_E
614 CFE_Wind_N,CFE_Wind_E,CFE_Wind_S,CFE_Wind_W = WindResCosFun(Vs_E,Axv,rho_air,
615     wind_u,wind_v,Cair_extend,delta_T)
616 CFE_m=TotalResCosFun(Vs_E,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,
617     rho_sea,nu_sea,g,delta_T)
618 CFE_Wave_N,CFE_Wave_E,CFE_Wave_S,CFE_Wave_W=WaveResCosFun(Vs_E,Lbwl,B,rho_sea
619     ,wave_d,wave_h,delta_T)
620 CFE_N=CFE_Wind_N+CFE_Wave_N+CFE_m
621 CFE_E=CFE_Wind_E+CFE_Wave_E+CFE_m
622 CFE_S=CFE_Wind_S+CFE_Wave_S+CFE_m
623 CFE_W=CFE_Wind_W+CFE_Wave_W+CFE_m
624 show = True
625 if show:
626     plt.plot(ox, oy, '.k')
627     plt.plot(sx, sy, 'og')
628     plt.plot(gx, gy, 'or')
629     # plt.grid('True')
630     plt.axis('equal')
631     # plt.show()
632
633 """get optimum path for this optimum velocity"""
634 rx, ry, TotalCost_min = dijkstra.planning(sx, sy, gx, gy,CFE_N,CFE_E,CFE_S,
635     CFE_W)
636 if show:
637     print('Find the effective path' )
638     # print('The efficient velocity is [m/s]:',Vs_E)
639     print('The efficient velocity is [knot]:',Vs_E/0.5144)
640     # print('The minimum cost is [J]:',TotalCost_min)
641     print('The minimum cost is [MJ]:',TotalCost_min/1000000)
642     plt.plot(rx, ry, '-r')
643     plt.pause(0.01)
644     plt.show()

```

Listing A.10: Testing codes (section 4.2)