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

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 whithin 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.

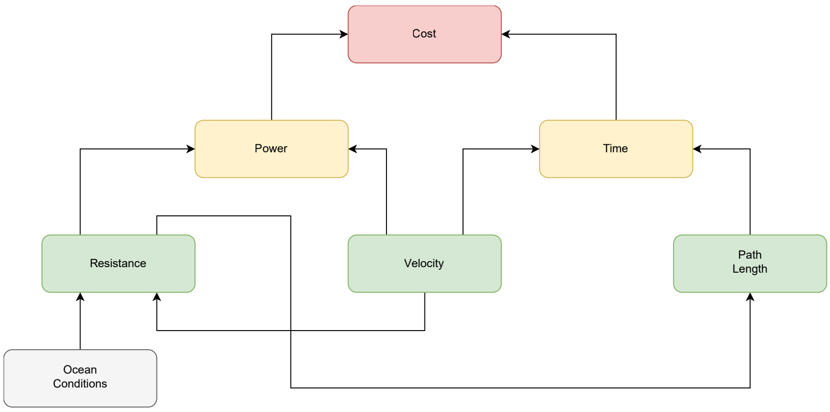


Figure 1.1: Cost analysis

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:

$$C = \int_0^t P(t)dt \tag{2.1}$$

- 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:

$$P_E = T_e * V_G = R * V_G \tag{2.2}$$

- 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:

$$P_E = \eta P \tag{2.3}$$

- 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:

$$t = L_{path}/V_G \tag{2.4}$$

- 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

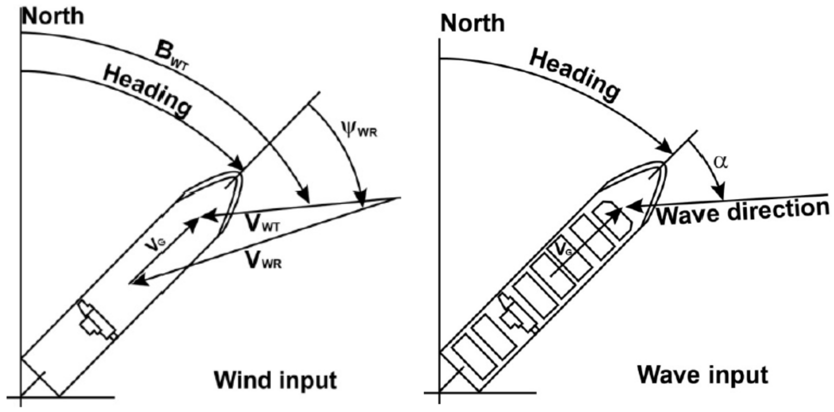
Ship is a kind of vehicle travelling between water and air, which means that the resistance of shipping could be classified as two parts based on different causes. The resistance R_{wind} which caused by wind could be calculated by the ITTC recommended method. In some places, this resistance R_{wind} 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:

$$R_{total} = R_{wind} + R_{hull} + R_{wave} \tag{2.5}$$

- 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:

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

- 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:

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

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

$$V_{\mathrm{WRref}} = \sqrt{V_{\mathrm{WTref}}^2 + V_{\mathrm{G}}^2 + 2V_{\mathrm{WTref}} V_{\mathrm{G}} \cos\left(\psi_{\mathrm{WT}} - \psi\right)}$$

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

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

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:

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

- 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::

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

- 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:

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

- 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:

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

- 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:

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

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

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

k_L in Eq.(2.10) is defined as:

$$k_L = e_1 \cdot L^{e2} \tag{2.14}$$

These formulas are valid when Froudes number in following intervals:

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

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

$$R_{Tmax} = h_1 \cdot R_{\mathrm{Tmean}} \tag{2.16}$$

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:

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

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:

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

- $H_{\mathrm{W1/3}}$: 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

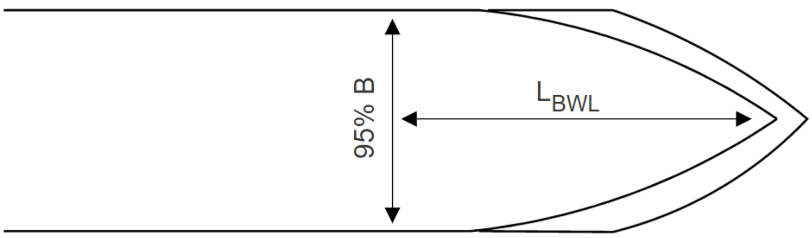


Figure 2.2: Definition of L_{BWL}

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 origianl 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 toatl 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:

```
create US,VS
while US is not empty:
    pick u_min in VS as u
    for each unvisited neighbour v of u:
        CC(v) = cost(u, v)
        if CC(v) < min cost(u, v):
            min cost(0,v) = CC(v)
    Ncost(v_min)=Ncost(u)+cost(u, v_min)
    if v_min == destination node:
        find EP
    else:
        removed v_min from US
        add v_min into VS
```

Listing 2.1: Pseudocode Dijkstra algorithm

- US: Unvisited set
- VS: Visited set

- $Ncost(u)$: Cost of the efficent 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, expect 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 trave 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 the vertical position of the anemometer, are same

(11) Type of this ship is consider 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:

```
read data and reconstruct data

for i in iterations:
    calculate velocity
    for each velocity:
        for each node:
            for each direction:
                calculate cost

        find effective path
        if time <= given time:
            record total cost
        else:
            could not complete task

if no velocity is eligible:
    ask for more iteration
else:
    compare total cost of different paths
    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 uses 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 trave 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:

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

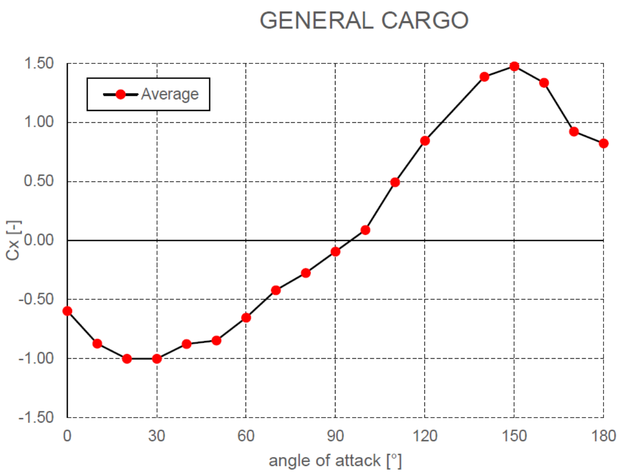
Although changing course would not increase cost, the heading angle of ship will still affect the resistance caused by wind and wave, in which case resistance needs to be calculated for each direction. This is the reason why limitations are put on directions. Owing to Assumption (2) and (7), we could 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:

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

$\Delta C_{East}(a,d)$, $\Delta C_{South}(a,e)$ 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:



Angle of attack [°]	Cx [-] Average
0	-0.60
10	-0.87
20	-1.00
30	-1.00
40	-0.88
50	-0.85
60	-0.65
70	-0.42
80	-0.27
90	-0.09
100	0.09
110	0.49
120	0.84
140	1.39
150	1.47
160	1.34
170	0.92
180	0.82

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

For convenience, $C_x(130°)$ 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:

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

where:

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

If: $\quad V_{\mathrm{G}}+V_{\mathrm{WT}} \cos \left(\psi_{\mathrm{WT}}-\psi\right) \geq 0$

$$\psi_{\mathrm{WR}}=\tan ^{-1} \frac{V_{\mathrm{WT}} \sin \left(\psi_{\mathrm{WT}}-\psi\right)}{V_{\mathrm{G}}+V_{\mathrm{WT}} \cos \left(\psi_{\mathrm{WT}}-\psi\right)}$$

Range: $-90^{\circ} \sim 90^{\circ}$

If: $\quad V_{\mathrm{G}}+V_{\mathrm{WT}} \cos \left(\psi_{\mathrm{WT}}-\psi\right)<0$

$$\psi_{\mathrm{WR}}=\tan ^{-1} \frac{V_{\mathrm{WT}} \sin \left(\psi_{\mathrm{WT}}-\psi\right)}{V_{\mathrm{G}}+V_{\mathrm{WT}} \cos \left(\psi_{\mathrm{WT}}-\psi\right)}+180^{\circ}$$

Range: $-90^{\circ} \sim 90^{\circ} \quad$ and $\quad 90^{\circ} \sim 270^{\circ}$

Then, if: $\quad \psi_{\mathrm{WR}}<0$

$$\psi_{\mathrm{WR}}=\psi_{\mathrm{WR}}+360^{\circ}$$

Range: $0^{\circ} \sim 90^{\circ} \quad$ and $\quad 90^{\circ} \sim 270^{\circ} \quad$ and $\quad 270^{\circ} \sim 360^{\circ}$

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):

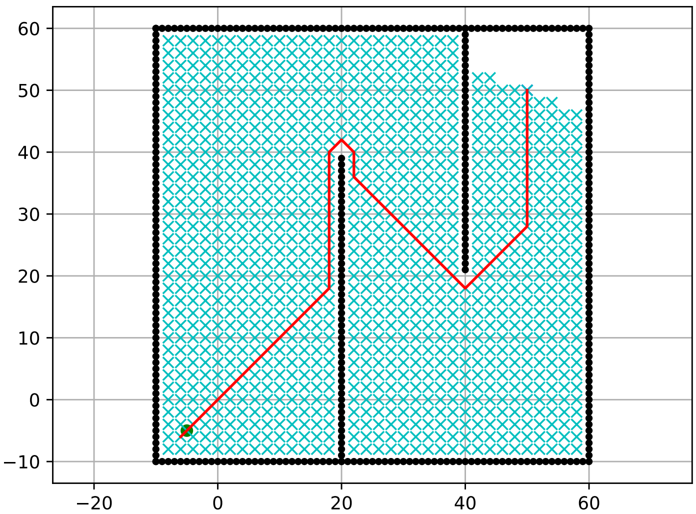


Figure 3.2: Grid based Dijkstra planning (Atsushi's project)

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:

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

- $C_{total}(V_{G,j})$: 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 defind as:

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

- 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 defind as:

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

- 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

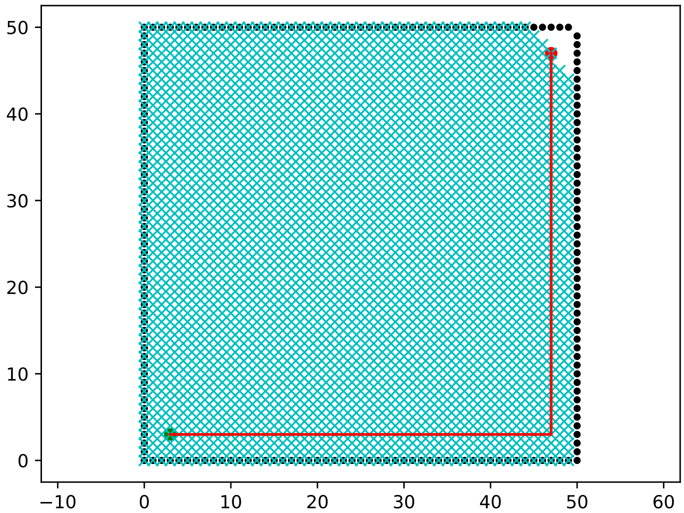


Figure 4.1: Result of Test 1

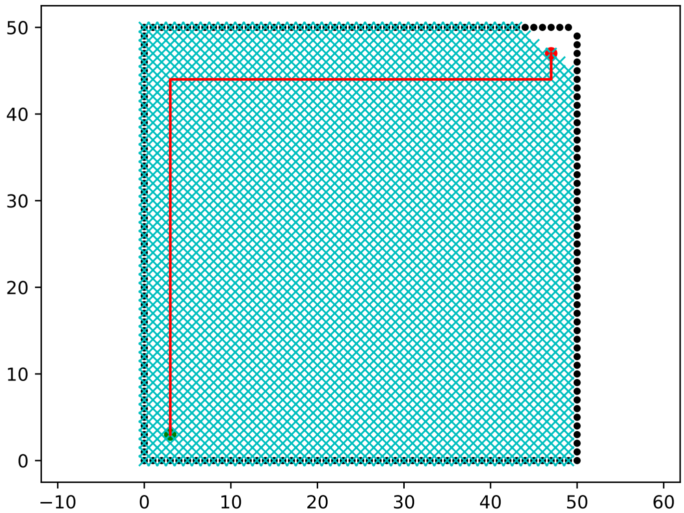


Figure 4.2: Result of Test 2

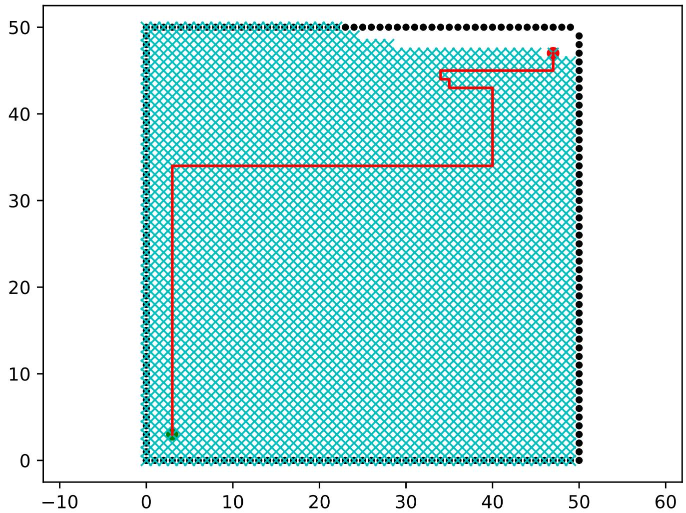


Figure 4.3: Result of Test 3

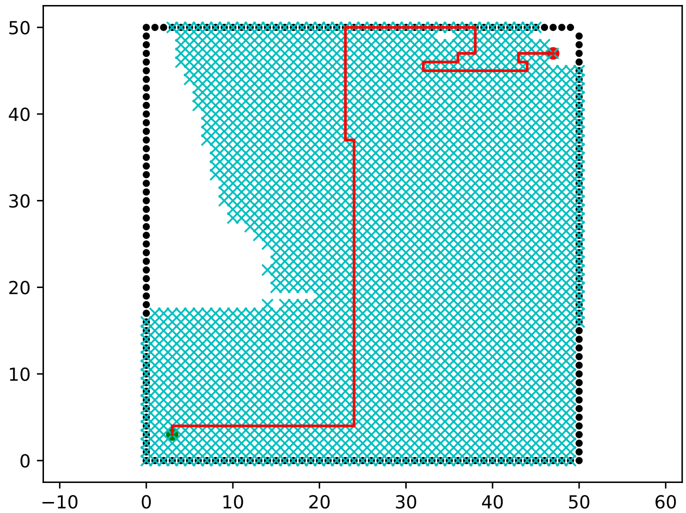


Figure 4.4: Result of Test 5

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

```
def ReadCsvWind(filepath,xcol,yrow):
    """
    Written by Weijian Yang, email: weijiany@stud.ntnu.no
    Function: read data & reconstruct data
    Parameters:
        filepath: filepath of CSV files (delimiter is ";")
        xcol: number of columns in grid
        yrow: number of rows in grid
    Return value:
        dataname: matrix of reconstructed data
    """
    file = open(filepath, "rb")
    filedata=np.loadtxt(file, delimiter=";")
    file.close()
    filearray=np.array(filedata)

    """ +2: just in case for that: data is not enough for grid """
    num_row=xcol+2
    num_col=yrow+2
    dataname = np.zeros(shape=(num_row,num_col))
    for i in range(num_row):
        for j in range(num_col):
            dataname[i,j]=filearray[i, j]
    return dataname
```

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

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

```
    file =pd.read_csv(filepath)

    """ +2: just in case for that: data is not enough for grid """
    num_row=xcol+2
    num_col=yrow+2
    dataname = file.values[0:num_row,0:num_col]
    return dataname
```

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

```
def WindResCosFun(Vs,Axv,rho_air,wind_u,wind_v,Cair_extend,dt):
    """
    Written by Weijian Yang, email: weijiany@stud.ntnu.no
    Function: calculate cost caused by wind
    Method: ITTC recommended method (2017)
    Parameters:
        Vs: measured ship's speed over ground
        Axv:  ship's area of maximum transverse section exposed to the wind
        rho_air: mass density of air
        wind_u: wind speed (north-south direction)
        wind_v: wind speed (east-west direction)
        Cair_extend: extended data sets of the wind resistance coefficient (0~360 degrees)
        dt: time needed for travelling between two neighbour nodes
    Return value:
        CF_Wind_North: cost caused by wind, if travelling to north neighbour nodes
        CF_Wind_East: cost caused by wind, if travelling to east neighbour nodes
        CF_Wind_South: cost caused by wind, if travelling to south neighbour nodes
        CF_Wind_West: cost caused by wind, if travelling to west neighbour nodes
    """
    angle_ship=np.mat([0,90,180,270])  #Angle of ship: North, East, South and West
    for k in range(angle_ship.shape[1]):
        vel_wind_squre=np.mat(np.zeros(wind_v.shape))
        vel_wt_squre=np.mat(np.zeros(wind_u.shape))
        angle_wt=np.mat(np.zeros(wind_u.shape))
        angle_wt_rel=np.mat(np.zeros(wind_v.shape))
        angle_rel=np.mat(np.zeros(wind_u.shape))
        Cx=np.mat(np.zeros(wind_v.shape))
        if k == 0:
            wind_u_north=wind_u+Vs
            for i in range(wind_u.shape[0]):
                for j in range(wind_v.shape[1]):
                    vel_wt_squre[i,j]=wind_u_north[i, j]**2+wind_v[i, j]**2 #Vwt: true wind velocity
                    if vel_wt_squre[i,j]==0:
                        angle_wt[i,j]=0
                    else:
                        angle_wt[i,j]=math.degrees(math.acos(wind_v[i,j]/math.sqrt(vel_wt_squre[i,j]))) #Awt:true wind direction

                    angle_wt_rel[i,j]=angle_wt[i,j]-angle_ship[0,k]+180 #Awt-Aship
                    vel_wind_squre[i,j]=vel_wt_squre[i,j]+Vs**2+math.sqrt(vel_wt_squre[i,j])*Vs*math.cos(math.radians(angle_wt_rel[i,j])) #V_WRef
                    nume=math.sqrt(vel_wt_squre[i,j])*math.sin(math.radians(angle_wt_rel[i,j]))
                    deno=Vs+math.sqrt(vel_wt_squre[i,j])*math.cos(math.radians(angle_wt_rel[i,j]))
```

```
                    if deno==0:
                        if nume<0:
                            angle_rel[i,j]=-90
                        if nume>0:
                            angle_rel[i,j]=90
                        if nume==0:
                            angle_rel[i,j]=0
                    elif deno<0:
                        angle_rel[i,j]=math.degrees(math.atan(nume/deno))+180 #A_WRef
                    else:
                        angle_rel[i,j]=math.degrees(math.atan(nume/deno))
                    if angle_rel[i,j]< 0:
                        angle_rel[i,j]=angle_rel[i,j]+360

                    weight=angle_rel[i,j]/10
                    index=int(angle_rel[i,j]//10)
                    Cx[i,j]=(weight-index)*Cair_extend[0,index+1]+(1-weight+index)*Cair_extend[0,index]
            Rwind_N=0.5*rho_air*Axv*(Cx*vel_wind_squre-Cair_extend[0,0]*Vs**2)
            CF_Wind_North=Rwind_N*Vs*dt

        if k == 1:
            wind_v_east=wind_v+Vs
            for i in range(wind_u.shape[0]):
                for j in range(wind_v.shape[1]):
                    vel_wt_squre[i,j]=wind_u[i, j]**2+wind_v_east[i, j]**2 #Vwt: true wind velocity
                    if vel_wt_squre[i,j]==0:
                        angle_wt[i,j]=0
                    else:
                    """
                    It is hard to shorten codes because equations for calculating true wind direction are different.
                    For comparison: (north) angle_wt[i,j]=math.degrees(math.acos(wind_v[i,j]/math.sqrt(vel_wt_squre[i,j])))
                    """
                        angle_wt[i,j]=math.degrees(math.asin(wind_v_east[i,j]/math.sqrt(vel_wt_squre[i,j]))) #Awt:true wind direction

    ...

    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)

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

```
        CF_Wave_S: cost caused by wave, if travelling to south neighbour nodes
        CF_Wave_W: cost caused by wave, if travelling to west neighbour nodes
    """
    angle_ship=np.mat([0,90,180,270])  #Angle of ship: North, East, South and
     West
    for k in range(angle_ship.shape[1]):
        angle_rel=np.mat(np.zeros(wave_d.shape))
        R_wave=np.mat(np.zeros(wave_h.shape))
        for i in range(wave_d.shape[0]):
            for j in range(wave_h.shape[1]):
                angle_rel[i,j]=wave_d[i,j]-angle_ship[0,k]
                while angle_rel[i,j]<-180 or angle_rel[i,j]>=180:
                    if angle_rel[i,j]< -180:
                        angle_rel[i,j]=angle_rel[i,j]+360
                    if angle_rel[i,j]>=180:
                        angle_rel[i,j]=angle_rel[i,j]-360

                if angle_rel[i,j]<-45 or angle_rel[i,j]>45:
                    R_wave[i,j]=0
                else:
                    R_wave[i,j]=rho_sea*g*B*math.sqrt(B/Lbwl)/16*wave_h[i,j]**2
                C_wave=R_wave*Vs*dt
        if k==0:
            CF_Wave_N=C_wave
        if k==1:
            CF_Wave_E=C_wave
        if k==2:
            CF_Wave_S=C_wave
        if k==0:
            CF_Wave_W=C_wave
    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)

```
"""
Given in class
Rewritten by Weijian Yang, email: weijiany@stud.ntnu.no
Function: calculate cost caused by the reaction between hull and calm water
Method: Hollenbach
"""
#input some data
#Vsvec,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr
Vsvec=float(input('Velocity of Ships(m/s):'))
L=float(input('Length of Ship(meters):'))
Lwl=float(input('Length of Water Line(meters):'))
Los=float(input('Length over Surface(meters):'))
B=float(input('Beam(meters):'))
TF=float(input('Draft of Fore Propeller(meters):'))
TA=float(input('Draft of Aft Propeller(meters):'))
CB=float(input('Block coefficient:'))
S=float(input('Wetted Surface(square meters):'))
Dp=float(input('Propeller diameter(meters):'))
NRud=float(input('Number of rudders:'))
NBrac=float(input('Number of brackets:'))
NBoss=float(input('Number of bossings:'))
NThr=float(input('Number of side thrusters:'))


#Calculation of 'Froude length', Lfn:
if Los/L < 1:
    Lfn = Los
elif (Los/L >= 1) & (Los/L < 1.1):
    Lfn = L+2/3*(Los-L)
```

```
elif Los/L >= 1.1:
    Lfn = 1.0667*L

# 'Mean' resistance coefficients
a = np.mat([-0.3382, 0.8086, -6.0258, -3.5632, 9.4405, 0.0146, 0, 0, 0, 0]) #a1 means a[0,0]
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]
d = np.mat([0.854, -1.228, 0.497])
e = np.mat([2.1701, -0.1602])
f  = np.mat([0.17, 0.20, 0.60])
g = np.mat([0.642, -0.635, 0.150])
#'Minimum' resistance coefficients
a_min = np.mat([-0.3382,0.8086,-6.0258,-3.5632,0,0,0,0,0,0])
b_min = np.mat
    ([[-0.91424,13.3893,90.5960],[4.6614,-39.721,-351.483],[-1.14215,-12.3296,459.254]])

d_min = np.mat([0,0,0])
e_min = np.mat([1,0])
f_min  = np.mat([0.17,0.2,0.6])
g_min = np.mat([0.614,-0.717,0.261])

cc = 0
# Loop over velocities
for i in range(Vsvec.size):
    Vs = Vsvec[0,i]

    cc = cc + 1

    Fn = Vs/((gravk*Lfn)**0.5) #Froude's number
    Fnkrit_help0=np.mat([1,CB,CB**2]) # Build Matrix for using transpose: Fnkrit_help0.T
    Fnkrit_help1 = d*Fnkrit_help0.T  # Fnkrit_help1=[[x]]   Matrix type
    Fnkrit=Fnkrit_help1[0,0]  # Fnkrit=x    Float type
    c1 = Fn/Fnkrit
    c1_min = Fn/Fnkrit
    Rns = Vs*L/nu #Reynold's number for ship
    if Rns == 0:  #Rns=0,log would get stuck
        CFs =0
    else :
        CFs = 0.075/(math.log10(Rns)-2)**2 #ITTC friction line for ship


    # Calculation of C_R for given ship
    # Mean value
    CRFnkrit = max(1.0,(Fn/Fnkrit)**c1)
    kL = e[0,0]*L**(e[0,1])

    # There is an error in the hollenbach paper and in Minsaas' 2003 textbook, which is corrected in this formula by dividing by 10
    CRstandard_help0=np.mat([1,Fn,Fn**2])
    CRstandard_help1=Fnkrit_help0*(b*CRstandard_help0.T)/10
    CRstandard=CRstandard_help1[0,0]

    #prod([T/B B/L Los/Lwl Lwl/L (1+(TA-TF)/L) Dp/TA (1+NRud) (1+NBrac) (1+NBoss) (1+NThr)].^a)
    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])
    prod_help1=np.mat(np.ones((1,10))) #build a Matrix[1,10]
    for j in range(a.size):  #prod_help=[prod_help[0,i].^a_min[0,i]]
            prod_help1[0,j]=prod_help[0,j]**a[0,j]
    prod_help2=np.prod(prod_help1,axis = 1) #prod function
```

```
    prod=prod_help2[0,0]

    CR_hollenbach = CRstandard*CRFnkrit*kL*prod
    CR = CR_hollenbach*B*T/S #Resistance coefficient, scaled for wetted surface
    C_Ts = CFs + CR  #Total resistance coeff. ship
    R_T_mean = C_Ts*rho/2*Vs**2*S #Total resistance to the ship

    #Minimum values

    #There is an error in the hollenbach paper and in Minsaas' 2003 textbook,
     which is corrected in this formula by dividing by 10
    CRstandard_min_help = Fnkrit_help0*(b_min*CRstandard_help0.T)/10
    CRstandard_min=CRstandard_min_help[0,0]
    #prod([T/B B/L Los/Lwl Lwl/L (1+(TA-TF)/L) Dp/TA (1+NRud) (1+NBrac) (1+NBoss)
      (1+NThr)].^a_min)
    prod_help_min1=np.mat(np.ones((1,10)))
    for j in range(a_min.size):                                              #
     prod_help=[prod_help[0,i].^a_min[0,i]]
            prod_help_min1[0,j]=prod_help[0,j]**a_min[0,j]
    prod_help_min2=np.prod(prod_help_min1,axis = 1)                 #prod
     function
    prod_min=prod_help_min2[0,0]

    CR_hollenbach_min = CRstandard_min*prod_min
    CR_min = CR_hollenbach_min*B*T/S


    # Total resistance coefficient of the ship
    C_Ts_min = CFs + CR_min
    # Total resistance
    R_T_min = C_Ts_min*rho/2*Vs**2*S
    #Propulsion power
    P_E_mean = R_T_mean * Vs       #[W]
    P_E_min = R_T_min * Vs             #[W]

    #print sth.
    # print('Vs =', Vs/0.5144,'knots')
    # print('Mean values')
    # print('CRh:',CR)
    # print('CF:',CFs)
    # print('CT:',C_Ts)
    # print('RT:',R_T_mean,'N')
    # print('Minimum values')
    # print('CRh:',CR_min)
    # print('CF:',CFs)
    # print('CT:',C_Ts_min)
    # print('RT:',R_T_min)

    # % Store results for plotting
    CFsvec = np.mat(np.zeros((1,Vsvec.size)))
    CRvec = np.mat(np.zeros((1,Vsvec.size)))
    C_Tsvec = np.mat(np.zeros((1,Vsvec.size)))
    R_T_meanvec = np.mat(np.zeros((1,Vsvec.size)))
    CR_minvec = np.mat(np.zeros((1,Vsvec.size)))
    C_Ts_minvec = np.mat(np.zeros((1,Vsvec.size)))
    R_T_minvec = np.mat(np.zeros((1,Vsvec.size)))
    P_E_meanvec = np.mat(np.zeros((1,Vsvec.size)))
    P_E_minvec = np.mat(np.zeros((1,Vsvec.size)))
    CFsvec[0,i] = CFs
    CRvec[0,i] = CR
    C_Tsvec[0,i] = C_Ts
    R_T_meanvec[0,i] = R_T_mean
```

```python
    CR_minvec[0,i] = CR_min
    C_Ts_minvec[0,i] = C_Ts_min
    R_T_minvec[0,i] = R_T_min
    P_E_meanvec[0,i] = P_E_mean
    P_E_minvec[0,i] = P_E_min

#This is the returned matrix
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)

```python
def TotalResCosFun(Vs,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,rho_sea,nu_sea,g,dt):
    """
    Modified by Weijian Yang, email: weijiany@stud.ntnu.no
    Function: calculate cost caused by the reaction between hull and calm water
    Method: Hollenbach
    Parameters:
        Vs: measured ship’s speed over ground
        L: length between perpendiculars
        Lwl: length of water line
        Los: length over surface
        B: beam
        TA: draught of aft propeller
        TF: draught of fore propeller
        CB: block coefficient
        S: wetted surface
        Dp: propeller diameter
        NRud: number of rudders
        NBrac: number of brackets
        NBoss: number of bossings
        NThr: number of side thrusters
        rho_sea: mass density of sea
        nu_sea: mass viscosity of sea
        g: gravitational acceleration
        dt: time needed for travelling between two neighbour nodes
    Return value:
        C_T: cost caused by the reaction between hull and calm water
    """
   rho = 1025         #Density of sea water [kg/m^3]
   gravk = 9.81       #Gravitational constant [m/s^2]
   nu = 1.1395E-6  #Viscosity of sea water [m/s^2]

   T = (TF+TA)/2
   """ Calculation of ’Froude length’, Lfn """
   if Los/L < 1:
      Lfn = Los
   elif (Los/L >= 1) & (Los/L < 1.1):
      Lfn = L+2/3*(Los-L)
   elif Los/L >= 1.1:
      Lfn = 1.0667*L

   # ’Mean’ resistance coefficients
   a = np.mat([-0.3382, 0.8086, -6.0258, -3.5632, 9.4405, 0.0146, 0, 0, 0, 0]) # a1 means a[0,0]
   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]
   d = np.mat([0.854, -1.228, 0.497])
   e = np.mat([2.1701, -0.1602])
   f  = np.mat([0.17, 0.20, 0.60])
```

```
    g = np.mat([0.642, -0.635, 0.150])

    Fn = Vs/((gravk*Lfn)**0.5) #Froude's number
    Fnkrit_help0=np.mat([1,CB,CB**2]) # Build Matrix for using transpose: Fnkrit_help0.T
    Fnkrit_help1 = d*Fnkrit_help0.T # Fnkrit_help1=[[x]] Matrix type
    Fnkrit=Fnkrit_help1[0,0] # Fnkrit=x  Float type
    c1 = Fn/Fnkrit
    Rns = Vs*L/nu #Reynold's number for ship
    if Rns == 0: #Rns=0,log would get stuck
       CFs =0
    else :
       CFs = 0.075/(math.log10(Rns)-2)**2 #ITTC friction line for ship

    """  Calculation of C_R for given ship  """
    # Mean value
    CRFnkrit = max(1.0,(Fn/Fnkrit)**c1)
    kL = e[0,0]*L**(e[0,1])

    # There is an error in the hollenbach paper and in Minsaas' 2003 textbook, which is corrected in this formula by dividing by 10
    CRstandard_help0=np.mat([1,Fn,Fn**2])
    CRstandard_help1=Fnkrit_help0*(b*CRstandard_help0.T)/10
    CRstandard=CRstandard_help1[0,0]

    #prod([T/B B/L Los/Lwl Lwl/L (1+(TA-TF)/L) Dp/TA (1+NRud) (1+NBrac) (1+NBoss) (1+NThr)].^a)
    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])
    prod_help1=np.mat(np.ones((1,10))) #build a Matrix[1,10]
    for j in range(a.size): #prod_help=[prod_help[0,i].^a_min[0,i]]
            prod_help1[0,j]=prod_help[0,j]**a[0,j]
    prod_help2=np.prod(prod_help1,axis = 1) #prod function
    prod=prod_help2[0,0]

    CR_hollenbach = CRstandard*CRFnkrit*kL*prod
    CR = CR_hollenbach*B*T/S #Resistance coefficient, scaled for wetted surface
    C_Ts = CFs + CR #Total resistance coeff. ship
    R_T_mean = C_Ts*rho/2*Vs**2*S #Total resistance to the ship

    Fn_min=min(f[0,0],f[0,0]+f[0,1]*(f[0,2]-CB))
    Fn_max=g[0,0]+g[0,1]*CB+g[0,2]*CB**3

    if Fn>Fn_max:
       #R_T=h1*R_T_mean
       R_T=1.204*R_T_mean  #h1=1.204
    elif Fn<Fn_min:
       #CRFnkrit=kL=1.0
       CR_min=CRstandard*prod*B*T/S
       R_T= (CFs + CR_min)*rho/2*Vs**2*S
    else:
       R_T=R_T_mean

    C_T = R_T*Vs*dt
    return C_T
```

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

```
"""
Grid based Dijkstra planning
author: Atsushi Sakai
```

```
link: https://github.com/AtsushiSakai/PythonRobotics/blob/master/PathPlanning/
    Dijkstra/dijkstra.py
"""

import matplotlib.pyplot as plt
import math

show_animation = True


class Dijkstra:

    def __init__(self, ox, oy, resolution, robot_radius):
        """
        Initialize map for a star planning
        ox: x position list of Obstacles [m]
        oy: y position list of Obstacles [m]
        resolution: grid resolution [m]
        rr: robot radius[m]
        """

        self.min_x = None
        self.min_y = None
        self.max_x = None
        self.max_y = None
        self.x_width = None
        self.y_width = None
        self.obstacle_map = None

        self.resolution = resolution
        self.robot_radius = robot_radius
        self.calc_obstacle_map(ox, oy)
        self.motion = self.get_motion_model()

    class Node:
        def __init__(self, x, y, cost, parent_index):
            self.x = x  # index of grid
            self.y = y  # index of grid
            self.cost = cost
            self.parent_index = parent_index  # index of previous Node

        def __str__(self):
            return str(self.x) + "," + str(self.y) + "," + str(
                self.cost) + "," + str(self.parent_index)

    def planning(self, sx, sy, gx, gy):
        """
        dijkstra path search
        input:
            s_x: start x position [m]
            s_y: start y position [m]
            gx: goal x position [m]
            gx: goal x position [m]
        output:
            rx: x position list of the final path
            ry: y position list of the final path
        """

        start_node = self.Node(self.calc_xy_index(sx, self.min_x),
                               self.calc_xy_index(sy, self.min_y), 0.0, -1)
        goal_node = self.Node(self.calc_xy_index(gx, self.min_x),
                              self.calc_xy_index(gy, self.min_y), 0.0, -1)
```

```python

        open_set, closed_set = dict(), dict()
        open_set[self.calc_index(start_node)] = start_node

        while 1:
            c_id = min(open_set, key=lambda o: open_set[o].cost)
            current = open_set[c_id]

            # show graph
            if show_animation:  # pragma: no cover
                plt.plot(self.calc_position(current.x, self.min_x),
                         self.calc_position(current.y, self.min_y), "xc")
                # for stopping simulation with the esc key.
                plt.gcf().canvas.mpl_connect(
                    'key_release_event',
                    lambda event: [exit(0) if event.key == 'escape' else None])
                if len(closed_set.keys()) % 10 == 0:
                    plt.pause(0.001)

            if current.x == goal_node.x and current.y == goal_node.y:
                print("Find goal")
                goal_node.parent_index = current.parent_index
                goal_node.cost = current.cost
                break

            # Remove the item from the open set
            del open_set[c_id]

            # Add it to the closed set
            closed_set[c_id] = current

            # expand search grid based on motion model
            for move_x, move_y, move_cost in self.motion:
                node = self.Node(current.x + move_x,
                                 current.y + move_y,
                                 current.cost + move_cost, c_id)
                n_id = self.calc_index(node)

                if n_id in closed_set:
                    continue

                if not self.verify_node(node):
                    continue

                if n_id not in open_set:
                    open_set[n_id] = node  # Discover a new node
                else:
                    if open_set[n_id].cost >= node.cost:
                        # This path is the best until now. record it!
                        open_set[n_id] = node

        rx, ry = self.calc_final_path(goal_node, closed_set)

        return rx, ry

    def calc_final_path(self, goal_node, closed_set):
        # generate final course
        rx, ry = [self.calc_position(goal_node.x, self.min_x)], [
            self.calc_position(goal_node.y, self.min_y)]
        parent_index = goal_node.parent_index
        while parent_index != -1:
            n = closed_set[parent_index]
```

```
            rx.append(self.calc_position(n.x, self.min_x))
            ry.append(self.calc_position(n.y, self.min_y))
            parent_index = n.parent_index

        return rx, ry

    def calc_position(self, index, minp):
        pos = index * self.resolution + minp
        return pos

    def calc_xy_index(self, position, minp):
        return round((position - minp) / self.resolution)

    def calc_index(self, node):
        return (node.y - self.min_y) * self.x_width + (node.x - self.min_x)

    def verify_node(self, node):
        px = self.calc_position(node.x, self.min_x)
        py = self.calc_position(node.y, self.min_y)

        if px < self.min_x:
            return False
        if py < self.min_y:
            return False
        if px >= self.max_x:
            return False
        if py >= self.max_y:
            return False

        if self.obstacle_map[node.x][node.y]:
            return False

        return True

    def calc_obstacle_map(self, ox, oy):

        self.min_x = round(min(ox))
        self.min_y = round(min(oy))
        self.max_x = round(max(ox))
        self.max_y = round(max(oy))
        print("min_x:", self.min_x)
        print("min_y:", self.min_y)
        print("max_x:", self.max_x)
        print("max_y:", self.max_y)

        self.x_width = round((self.max_x - self.min_x) / self.resolution)
        self.y_width = round((self.max_y - self.min_y) / self.resolution)
        print("x_width:", self.x_width)
        print("y_width:", self.y_width)

        # obstacle map generation
        self.obstacle_map = [[False for _ in range(self.y_width)]
                             for _ in range(self.x_width)]
        for ix in range(self.x_width):
            x = self.calc_position(ix, self.min_x)
            for iy in range(self.y_width):
                y = self.calc_position(iy, self.min_y)
                for iox, ioy in zip(ox, oy):
                    d = math.hypot(iox - x, ioy - y)
                    if d <= self.robot_radius:
                        self.obstacle_map[ix][iy] = True
                        break
```

```python

    @staticmethod
    def get_motion_model():
        # dx, dy, cost
        motion = [[1, 0, 1],
                  [0, 1, 1],
                  [-1, 0, 1],
                  [0, -1, 1],
                  [-1, -1, math.sqrt(2)],
                  [-1, 1, math.sqrt(2)],
                  [1, -1, math.sqrt(2)],
                  [1, 1, math.sqrt(2)]]

        return motion


def main():
    print(__file__ + " start!!")

    # start and goal position
    sx = -5.0  # [m]
    sy = -5.0  # [m]
    gx = 50.0  # [m]
    gy = 50.0  # [m]
    grid_size = 2.0  # [m]
    robot_radius = 1.0  # [m]

    # set obstacle positions
    ox, oy = [], []
    for i in range(-10, 60):
        ox.append(i)
        oy.append(-10.0)
    for i in range(-10, 60):
        ox.append(60.0)
        oy.append(i)
    for i in range(-10, 61):
        ox.append(i)
        oy.append(60.0)
    for i in range(-10, 61):
        ox.append(-10.0)
        oy.append(i)
    for i in range(-10, 40):
        ox.append(20.0)
        oy.append(i)
    for i in range(0, 40):
        ox.append(40.0)
        oy.append(60.0 - i)

    if show_animation:  # pragma: no cover
        plt.plot(ox, oy, ".k")
        plt.plot(sx, sy, "og")
        plt.plot(gx, gy, "xb")
        plt.grid(True)
        plt.axis("equal")

    dijkstra = Dijkstra(ox, oy, grid_size, robot_radius)
    rx, ry = dijkstra.planning(sx, sy, gx, gy)

    if show_animation:  # pragma: no cover
        plt.plot(rx, ry, "-r")
        plt.pause(0.01)
        plt.show()
```

```python


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

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

```python
class Dijkstra:
    """
    Modified by Weijian Yang, email: weijiany@stud.ntnu.no
    Function: find the efficient path for a given velocity
    Method: Dijkstra
    Parameters:
        ox: x position list of boundaries (Obstacles)
        oy: y position list of boundaries (Obstacles)
        sx: x position of the start point
        sy: y position of the start point
        gx: x position of the goal point
        gy: y position of the goal point
        CF_N: costs of movements in north direction
        CF_E: costs of movements in east direction
        CF_S: costs of movements in south direction
        CF_W: costs of movements in west direction
    Return values:
        rx: x position list of current path
        ry: y position list of current path
        goal_node.cost: total cost of current path
    """
    def __init__(self, ox, oy):
        # initialize parameters
        self.min_x = None
        self.max_x = None
        self.min_y = None
        self.max_y = None
        self.x_grid_num = None
        self.y_grid_num = None
        self.obstacle_map = None

        self.calc_obstacle_grid_map(ox, oy)

    def calc_obstacle_grid_map(self, ox, oy):
        """ build obstacle map """
        """
            Parameters:
                ox: x position list of boundaries (Obstacles)
                oy: y position list of boundaries (Obstacles)
        """
        # 1. get boundaries' values of the environment
        self.min_x = round(min(ox))
        self.max_x = round(max(ox))
        self.min_y = round(min(oy))
        self.max_y = round(max(oy))

        # 2. calculate needed numbers of x,y in map
        self.x_grid_num = round(self.max_x - self.min_x)
        self.y_grid_num = round(self.max_y - self.min_y)

        # 3. obstacle map generation
        self.obstacle_map = [[False for _ in range(self.x_grid_num)] for _ in
 range(self.y_grid_num)]

    def planning(self, sx, sy, gx, gy ,CF_N,CF_E,CF_S,CF_W):
        """ dijkstra path search """
```

```
        """
            Parameters:
                sx: x position of the start point
                sy: y position of the start point
                gx: x position of the goal point
                gy: y position of the goal point
                CF_N: costs of movements in north direction
                CF_E: costs of movements in east direction
                CF_S: costs of movements in south direction
                CF_W: costs of movements in west direction
            Return values:
                rx: x position list of current path
                ry: y position list of current path
                goal_node.cost: total cost of current path
            """
        # 1. get start_node, goal_node
        sx_index = self.calc_xy_index(sx, self.min_x)
        sy_index = self.calc_xy_index(sy, self.min_y)
        gx_index = self.calc_xy_index(gx, self.min_x)
        gy_index = self.calc_xy_index(gy, self.min_y)
        start_node = self.Node(sx_index, sy_index, 0.0, -1)
        goal_node = self.Node(gx_index, gy_index, 0.0, -1)

        # 2. initialize open_set, close_set, put start_node into open_set
        open_set, close_set = dict(), dict()
        open_set[self.calc_index(start_node)] = start_node

        # 3. search
        while True:
            # (1). choose the node whose cost is minimum in open_set
            c_id = min(open_set, key=lambda o: open_set[o].cost)
            current = open_set[c_id]

            if show:
                plt.plot(self.calc_position(current.x, self.min_x),
                            self.calc_position(current.y, self.min_y), "xc")
                # for stopping simulation with the esc key.
                # plt.gcf().canvas.mpl_connect(
                #     'key_release_event',
                #     lambda event: [exit(0) if event.key == 'escape' else None])
                if len(close_set.keys()) % 10 == 0:
                    plt.pause(0.001)

            # (2). determine whether the current node is the end point
            if current.x == goal_node.x and current.y == goal_node.y:
                goal_node.parent_index = current.parent_index
                goal_node.cost = current.cost
                break

            # (3). remove the current node from the open set, add it to the
     closed set
            del open_set[c_id]
            close_set[c_id] = current

            # (4). expand search grid based on motion model
            self.robot_motion = self.get_motion_model(current.x,current.y,CF_N,
     CF_E,CF_S,CF_W)
            for move_x, move_y, move_cost in self.robot_motion:
                node = self.Node(current.x + move_x,
                                    current.y + move_y,
                                    current.cost + move_cost, c_id)
                n_id = self.calc_index(node)
```

```

                if n_id in close_set:
                    continue

                if not self.verify_node(node):
                    continue

                if n_id not in open_set:
                    open_set[n_id] = node   # discover a new node
                else:
                    if open_set[n_id].cost >= node.cost:
                        # This path is the best until now. record it!
                        open_set[n_id] = node

        rx, ry = self.calc_final_path(goal_node, close_set)

        return rx, ry, goal_node.cost

    def calc_final_path(self, goal_node, close_set):
        """ generate final course """
        rx = [self.calc_position(goal_node.x, self.min_x)]
        ry = [self.calc_position(goal_node.y, self.min_y)]

        parent_index = goal_node.parent_index
        while parent_index != -1:
            n = close_set[parent_index]
            rx.append(self.calc_position(n.x, self.min_x))
            ry.append(self.calc_position(n.y, self.min_y))
            parent_index = n.parent_index

        return rx, ry

    class Node:
        def __init__(self, x, y, cost,parent_index):
            self.x = x
            self.y = y
            self.cost = cost       # g(n)
            self.parent_index = parent_index
        #
        # def __str__(self):
        #     return str(self.x) + "," + str(self.y) + "," + str(self.cost) + "," + str(self.parent_index)
    """
    These 3 functions calc_index, calc_xy_index and calc_position are used for coordinate system transformation
    However, in this project, they are useless because values of x and y in grid map coordinates and xy map are the same.
    """
    def calc_index(self, node):
        index = node.y * self.x_grid_num + node.x
        return index

    def calc_xy_index(self, pos, min_p):
        index = round(pos - min_p)
        return index

    def calc_position(self, index, min_p):
        pos = min_p + index
        return pos

    def verify_node(self, node):
        """ check whether the current position is appropriate """
```

```python
        px = self.calc_position(node.x, self.min_x)
        py = self.calc_position(node.y, self.min_y)

        if px < self.min_x or px > self.max_x:
            return False
        if py < self.min_x or py > self.max_y:
            return False

        return True

    @staticmethod
    def get_motion_model(x,y,CF_N,CF_E,CF_S,CF_W):
        # dx, dy, cost
        data_x=x
        data_y=y
        model = [
            [0, 1, CF_N[data_x,data_y+1]],         # North
            [0, -1,CF_S[data_x,data_y-1]],          # South
            [-1, 0,CF_E[data_x-1,data_y]],          # East
            [1, 0, CF_W[data_x+1,data_y]],        # West
        ]
        return model
```

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

```python
...
""" Build boundary """
ox, oy = [], []
for i in range(0, xcol):        #north boundary
    ox.append(i)
    oy.append(xcol)
for i in range(0, yrow):         #east boundary
    ox.append(0)
    oy.append(i)
for i in range(0, xcol):        #south boundary
    ox.append(i)
    oy.append(0)
for i in range(0, yrow):         #west boundary
    ox.append(yrow)
    oy.append(i)
dijkstra = Dijkstra(ox, oy)

T_cost=np.mat(np.zeros((1,num_iter)))
TotalCost=T_cost
show = False
for i in range(num_iter):
    length=(abs(sx-gx)+abs(sy-gy))+i
    """
    find minimum cost in each velocity
    Vs_min=(abs(sx-gx)+abs(sy-gy))/t
    Vs_max=xcol*yrow/t
    """
    Vs=float(length*dist/t)
    delta_T=dist/Vs

    """calculate cost for each given velocity"""
    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)
    CF_m=TotalResCosFun(Vs,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,
     rho_sea,nu_sea,g,delta_T)
    CF_Wave_N,CF_Wave_E,CF_Wave_S,CF_Wave_W=WaveResCosFun(Vs,Lbwl,B,rho_sea,
```

```
wave_d,wave_h,delta_T)
    CF_N=CF_Wind_N+CF_Wave_N+CF_m
    CF_E=CF_Wind_E+CF_Wave_E+CF_m
    CF_S=CF_Wind_S+CF_Wave_S+CF_m
    CF_W=CF_Wind_W+CF_Wave_W+CF_m

    """find an efficient path for each given velocity"""
    rx, ry, T_cost[0,i] = dijkstra.planning(sx, sy, gx, gy,CF_N,CF_E,CF_S,CF_W)

    """check task time"""
    if len(rx)*delta_T>t:
        # print('The ship cannot reach the goal point in given time with this speed',Vs ,' [m/s]')
        print('The ship cannot reach the goal point in given time with this speed',Vs/0.5144 ,' [knot]')
        TotalCost[0,i]=0
    else:
        # print('The ship could reach the goal point in given time with this speed',Vs ,' [m/s]')
        print('The ship could reach the goal point in given time with this speed',Vs/0.5144 ,' [knot]')
        TotalCost[0,i]=T_cost[0,i]
if np.max(TotalCost)==0:
    print('The effective path could not be found after ',num_iter,' iterations' )
else:

    """compare total costs of paths"""
    minx,miny= np.where(TotalCost == np.min(TotalCost[np.nonzero(TotalCost)]))

    """get optimum velocity"""
    length_E=(abs(sx-gx)+abs(sy-gy))+miny
    Vs_E=float(length_E*dist/t)
    delta_TE=dist/Vs_E
    CFE_Wind_N,CFE_Wind_E,CFE_Wind_S,CFE_Wind_W = WindResCosFun(Vs_E,Axv,rho_air,wind_u,wind_v,Cair_extend,delta_T)
    CFE_m=TotalResCosFun(Vs_E,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,rho_sea,nu_sea,g,delta_T)
    CFE_Wave_N,CFE_Wave_E,CFE_Wave_S,CFE_Wave_W=WaveResCosFun(Vs_E,Lbwl,B,rho_sea,wave_d,wave_h,delta_T)
    CFE_N=CFE_Wind_N+CFE_Wave_N+CFE_m
    CFE_E=CFE_Wind_E+CFE_Wave_E+CFE_m
    CFE_S=CFE_Wind_S+CFE_Wave_S+CFE_m
    CFE_W=CFE_Wind_W+CFE_Wave_W+CFE_m
    show = True
    if show:
        plt.plot(ox, oy, '.k')
        plt.plot(sx, sy, 'og')
        plt.plot(gx, gy, 'or')
        # plt.grid('True')
        plt.axis('equal')
        # plt.show()

    """get optimum path for this optimum velocity"""
    rx, ry, TotalCost_min = dijkstra.planning(sx, sy, gx, gy,CFE_N,CFE_E,CFE_S,CFE_W)
    if show:
        print('Find the effective path' )
...
```

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

A.2 Testing codes

```
"""
Program tests
Course: TMR 4345
Project: Ship Routing
Author: Weijian Yang, email: weijiany@stud.ntnu.no
"""
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
## Function for importing data from Csv file
## numpy -> delimiter = ";"
def ReadCsvWind(filepath,xcol,yrow):
    file = open(filepath, "rb")
    filedata=np.loadtxt(file, delimiter=";")
    file.close()
    filearray=np.array(filedata)
    # +2: just in case for that: data is not enough for grid
    num_row=xcol+2
    num_col=yrow+2
    dataname = np.zeros(shape=(num_row,num_col))
    for i in range(num_row):
        for j in range(num_col):
            dataname[i,j]=filearray[i, j]
    return dataname
## pandas -> delimiter = "rows & columns"
def ReadCsvWave(filepath,xcol,yrow):
    file =pd.read_csv(filepath)
    # +2: just in case for that: data is not enough for grid
    num_row=xcol+2
    num_col=yrow+2
    dataname = file.values[0:num_row,0:num_col]
    return dataname
## Function for calculating wind resistance:
## R=0.5*rho_air*Avx*[Cair(relative wind angle)*V(relative wind velocity)^2-Cair(0)*V(ship velocity)^2]
def WindResCosFun(Vs,Axv,rho_air,wind_u,wind_v,Cair_extend,dt):
    angle_ship=np.mat([0,90,180,270])  #Angle of ship: North, East, South and West
    for k in range(angle_ship.shape[1]):
        vel_wind_squre=np.mat(np.zeros(wind_v.shape))
        vel_wt_squre=np.mat(np.zeros(wind_u.shape))
        angle_wt=np.mat(np.zeros(wind_u.shape))
        angle_wt_rel=np.mat(np.zeros(wind_v.shape))
        angle_rel=np.mat(np.zeros(wind_u.shape))
        Cx=np.mat(np.zeros(wind_v.shape))
        if k == 0:
            wind_u_north=wind_u+Vs
            for i in range(wind_u.shape[0]):
                for j in range(wind_v.shape[1]):
                    vel_wt_squre[i,j]=wind_u_north[i, j]**2+wind_v[i, j]**2     # Vwt:true wind velocity
                    if vel_wt_squre[i,j]==0:
                        angle_wt[i,j]=0
                    else:
                        angle_wt[i,j]=math.degrees(math.acos(wind_v[i,j]/math.sqrt(vel_wt_squre[i,j]))) #Awt:true wind direction

                    angle_wt_rel[i,j]=angle_wt[i,j]-angle_ship[0,k]+180         # Awt-Aship
```

```
                vel_wind_squre[i,j]=vel_wt_squre[i,j]+Vs**2+math.sqrt(vel_wt_squre[i,j])*Vs*math.cos(math.radians(angle_wt_rel[i,j])) #V_WRef
                nume=math.sqrt(vel_wt_squre[i,j])*math.sin(math.radians(angle_wt_rel[i,j]))
                deno=Vs+math.sqrt(vel_wt_squre[i,j])*math.cos(math.radians(angle_wt_rel[i,j]))
                if deno==0:
                    if nume<0:
                        angle_rel[i,j]=-90
                    if nume>0:
                        angle_rel[i,j]=90
                    if nume==0:
                        angle_rel[i,j]=0
                elif deno<0:
                    angle_rel[i,j]=math.degrees(math.atan(nume/deno))+180   #A_WRef
                else:
                    angle_rel[i,j]=math.degrees(math.atan(nume/deno))
                if angle_rel[i,j]< 0:
                    angle_rel[i,j]=angle_rel[i,j]+360

                weight=angle_rel[i,j]/10
                index=int(angle_rel[i,j]//10)
                Cx[i,j]=(weight-index)*Cair_extend[0,index+1]+(1-weight+index)*Cair_extend[0,index]
        Rwind_N=0.5*rho_air*Axv*(Cx*vel_wind_squre-Cair_extend[0,0]*Vs**2)
        CF_Wind_North=Rwind_N*Vs*dt

    if k == 1:
        wind_v_east=wind_v+Vs
        for i in range(wind_u.shape[0]):
            for j in range(wind_v.shape[1]):
                vel_wt_squre[i,j]=wind_u[i, j]**2+wind_v_east[i, j]**2     #Vwt:true wind velocity
                if vel_wt_squre[i,j]==0:
                    angle_wt[i,j]=0
                else:
                    angle_wt[i,j]=math.degrees(math.asin(wind_v_east[i,j]/math.sqrt(vel_wt_squre[i,j]))) #Awt:true wind direction

                angle_wt_rel[i,j]=angle_wt[i,j]-angle_ship[0,k]+180         #Awt-Aship
                vel_wind_squre[i,j]=vel_wt_squre[i,j]+Vs**2+math.sqrt(vel_wt_squre[i,j])*Vs*math.cos(math.radians(angle_wt_rel[i,j])) #V_WRef
                nume=math.sqrt(vel_wt_squre[i,j])*math.sin(math.radians(angle_wt_rel[i,j]))
                deno=Vs+math.sqrt(vel_wt_squre[i,j])*math.cos(math.radians(angle_wt_rel[i,j]))
                if deno==0:
                    if nume<0:
                        angle_rel[i,j]=-90
                    if nume>0:
                        angle_rel[i,j]=90
                    if nume==0:
                        angle_rel[i,j]=0
                elif deno<0:
                    angle_rel[i,j]=math.degrees(math.atan(nume/deno))+180   #A_WRef
                else:
                    angle_rel[i,j]=math.degrees(math.atan(nume/deno))
                if angle_rel[i,j]< 0:
                    angle_rel[i,j]=angle_rel[i,j]+360
```

```

                weight=angle_rel[i,j]/10
                index=int(angle_rel[i,j]//10)
                Cx[i,j]=(weight-index)*Cair_extend[0,index+1]+(1-weight+index)*Cair_extend[0,index]
        Rwind_E=0.5*rho_air*Axv*(Cx*vel_wind_squre-Cair_extend[0,0]*Vs**2)
        CF_Wind_East=Rwind_E*Vs*dt

    if k == 2:
        wind_u_south=wind_u-Vs
        for i in range(wind_u.shape[0]):
            for j in range(wind_v.shape[1]):
                vel_wt_squre[i,j]=wind_u_south[i,j]**2+wind_v[i,j]**2     #Vwt:true wind velocity
                if vel_wt_squre[i,j]==0:
                    angle_wt[i,j]=0
                else:
                    angle_wt[i,j]=math.degrees(math.acos(wind_v[i,j]/math.sqrt(vel_wt_squre[i,j]))) #Awt:true wind direction

                angle_wt_rel[i,j]=angle_wt[i,j]-angle_ship[0,k]+180         #Awt-Aship
                vel_wind_squre[i,j]=vel_wt_squre[i,j]+Vs**2+math.sqrt(vel_wt_squre[i,j])*Vs*math.cos(math.radians(angle_wt_rel[i,j])) #V_WRef
                nume=math.sqrt(vel_wt_squre[i,j])*math.sin(math.radians(angle_wt_rel[i,j]))
                deno=Vs+math.sqrt(vel_wt_squre[i,j])*math.cos(math.radians(angle_wt_rel[i,j]))
                if deno==0:
                    if nume<0:
                        angle_rel[i,j]=-90
                    if nume>0:
                        angle_rel[i,j]=90
                    if nume==0:
                        angle_rel[i,j]=0
                elif deno<0:
                    angle_rel[i,j]=math.degrees(math.atan(nume/deno))+180    #A_WRef
                else:
                    angle_rel[i,j]=math.degrees(math.atan(nume/deno))
                if angle_rel[i,j]< 0:
                    angle_rel[i,j]=angle_rel[i,j]+360

                weight=angle_rel[i,j]/10
                index=int(angle_rel[i,j]//10)
                Cx[i,j]=(weight-index)*Cair_extend[0,index+1]+(1-weight+index)*Cair_extend[0,index]
        Rwind_S=0.5*rho_air*Axv*(Cx*vel_wind_squre-Cair_extend[0,0]*Vs**2)
        CF_Wind_South=Rwind_S*Vs*dt

    if k == 3:
        wind_v_west=wind_v-Vs
        for i in range(wind_u.shape[0]):
            for j in range(wind_v.shape[1]):
                vel_wt_squre[i,j]=wind_u[i, j]**2+wind_v_west[i, j]**2     #Vwt:true wind velocity
                if vel_wt_squre[i,j]==0:
                    angle_wt[i,j]=0
                else:
                    angle_wt[i,j]=math.degrees(math.asin(wind_v_west[i,j]/math.sqrt(vel_wt_squre[i,j]))) #Awt:true wind direction

```

```
                    angle_wt_rel[i,j]=angle_wt[i,j]-angle_ship[0,k]+180         # Awt-Aship
                    vel_wind_squre[i,j]=vel_wt_squre[i,j]+Vs**2+math.sqrt(vel_wt_squre[i,j])*Vs*math.cos(math.radians(angle_wt_rel[i,j])) #V_WRef
                    nume=math.sqrt(vel_wt_squre[i,j])*math.sin(math.radians(angle_wt_rel[i,j]))
                    deno=Vs+math.sqrt(vel_wt_squre[i,j])*math.cos(math.radians(angle_wt_rel[i,j]))
                    if deno==0:
                        if nume<0:
                            angle_rel[i,j]=-90
                        if nume>0:
                            angle_rel[i,j]=90
                        if nume==0:
                            angle_rel[i,j]=0
                    elif deno<0:
                        angle_rel[i,j]=math.degrees(math.atan(nume/deno))+180       #A_WRef
                    else:
                        angle_rel[i,j]=math.degrees(math.atan(nume/deno))
                    if angle_rel[i,j]< 0:
                        angle_rel[i,j]=angle_rel[i,j]+360

                    weight=angle_rel[i,j]/10
                    index=int(angle_rel[i,j]//10)
                    Cx[i,j]=(weight-index)*Cair_extend[0,index+1]+(1-weight+index)*Cair_extend[0,index]
            Rwind_W=0.5*rho_air*Axv*(Cx*vel_wind_squre-Cair_extend[0,0]*Vs**2)
            CF_Wind_West=Rwind_W*Vs*dt

    return CF_Wind_North,CF_Wind_East,CF_Wind_South,CF_Wind_West
## Function for calculating total resistance (only constant velocity)
## Hollenbach Method (only return mean resistance)
def TotalResCosFun(Vs,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,rho_sea,nu_sea,g,dt):
    rho = 1025          #Density of sea water [kg/m^3]
    gravk = 9.81        #Gravitational constant [m/s^2]
    nu = 1.1395E-6  #Viscosity of sea water [m/s^2]

    T = (TF+TA)/2
    """ Calculation of 'Froude length', Lfn """
    if Los/L < 1:
        Lfn = Los
    elif (Los/L >= 1) & (Los/L < 1.1):
        Lfn = L+2/3*(Los-L)
    elif Los/L >= 1.1:
        Lfn = 1.0667*L

    # 'Mean' resistance coefficients
    a = np.mat([-0.3382, 0.8086, -6.0258, -3.5632, 9.4405, 0.0146, 0, 0, 0, 0])                                         #a1 means a[0,0]
    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]
    d = np.mat([0.854, -1.228, 0.497])
    e = np.mat([2.1701, -0.1602])
    f  = np.mat([0.17, 0.20, 0.60])
    g = np.mat([0.642, -0.635, 0.150])

    Fn = Vs/((gravk*Lfn)**0.5)                      #Froude's number
    Fnkrit_help0=np.mat([1,CB,CB**2])          # Build Matrix for using transpose : Fnkrit_help0.T
    Fnkrit_help1 = d*Fnkrit_help0.T              # Fnkrit_help1=[[x]]   Matrix
```

```
type
    Fnkrit=Fnkrit_help1[0,0]                           # Fnkrit=x Float type
    c1 = Fn/Fnkrit
    Rns = Vs*L/nu                                     #Reynold's number for ship
    if Rns == 0:                                                    #Rns=0,log would get stuck
        CFs =0
    else :
        CFs = 0.075/(math.log10(Rns)-2)**2      #ITTC friction line for ship


    """ Calculation of C_R for given ship"""
    # Mean value
    CRFnkrit = max(1.0,(Fn/Fnkrit)**c1)
    kL = e[0,0]*L**(e[0,1])

    # There is an error in the hollenbach paper and in Minsaas' 2003 textbook, which is corrected in this formula by dividing by 10
    CRstandard_help0=np.mat([1,Fn,Fn**2])
    CRstandard_help1=Fnkrit_help0*(b*CRstandard_help0.T)/10
    CRstandard=CRstandard_help1[0,0]

    #prod([T/B B/L Los/Lwl Lwl/L (1+(TA-TF)/L) Dp/TA (1+NRud) (1+NBrac) (1+NBoss) (1+NThr)].^a)
    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])
    prod_help1=np.mat(np.ones((1,10)))                           #build a Matrix [1,10]
    for j in range(a.size):                                            #prod_help =[prod_help[0,i].^a_min[0,i]]
            prod_help1[0,j]=prod_help[0,j]**a[0,j]
    prod_help2=np.prod(prod_help1,axis = 1)                   #prod function
    prod=prod_help2[0,0]

    CR_hollenbach = CRstandard*CRFnkrit*kL*prod
    CR = CR_hollenbach*B*T/S               #Resistance coefficient, scaled for wetted surface
    C_Ts = CFs + CR                                    #Total resistance coeff. ship
    R_T_mean = C_Ts*rho/2*Vs**2*S      #Total resistance to the ship

    Fn_min=min(f[0,0],f[0,0]+f[0,1]*(f[0,2]-CB))
    Fn_max=g[0,0]+g[0,1]*CB+g[0,2]*CB**3

    if Fn>Fn_max:
        #R_T=h1*R_T_mean
        R_T=1.204*R_T_mean  #h1=1.204
    elif Fn<Fn_min:
        #CRFnkrit=kL=1.0
        CR_min=CRstandard*prod*B*T/S
        R_T= (CFs + CR_min)*rho/2*Vs**2*S
    else:
        R_T=R_T_mean

    C_T = R_T*Vs*dt
    return C_T
## Function for calculating added resistance (-45 degrees ~ 45 degrees)
## R=1/16*rho_sea*g*H^2*B*sqrt(B/Lbwl)
def WaveResCosFun(Vs,Lbwl,B,rho_sea,wave_d,wave_h,dt):
    angle_ship=np.mat([0,90,180,270])  #Angle of ship: North, East, South and West
```

```
    for k in range(angle_ship.shape[1]):
        angle_rel=np.mat(np.zeros(wave_d.shape))
        R_wave=np.mat(np.zeros(wave_h.shape))
        for i in range(wave_d.shape[0]):
            for j in range(wave_h.shape[1]):
                angle_rel[i,j]=wave_d[i,j]-angle_ship[0,k]
                while angle_rel[i,j]<-180 or angle_rel[i,j]>=180:
                    if angle_rel[i,j]< -180:
                        angle_rel[i,j]=angle_rel[i,j]+360
                    if angle_rel[i,j]>=180:
                        angle_rel[i,j]=angle_rel[i,j]-360

                if angle_rel[i,j]<-45 or angle_rel[i,j]>45:
                    R_wave[i,j]=0
                else:
                    R_wave[i,j]=rho_sea*g*B*math.sqrt(B/Lbwl)/16*wave_h[i,j]**2
                C_wave=R_wave*Vs*dt
        if k==0:
            CF_Wave_N=C_wave
        if k==1:
            CF_Wave_E=C_wave
        if k==2:
            CF_Wave_S=C_wave
        if k==0:
            CF_Wave_W=C_wave
    return CF_Wave_N,CF_Wave_E,CF_Wave_S,CF_Wave_W
## Dijkstra's alogrithm (find path)
class Dijkstra:
    def __init__(self, ox, oy):
        # initialize parameters
        self.min_x = None
        self.max_x = None
        self.min_y = None
        self.max_y = None
        self.x_grid_num = None
        self.y_grid_num = None
        self.obstacle_map = None
        self.calc_obstacle_grid_map(ox, oy)

    def calc_obstacle_grid_map(self, ox, oy):
        """ build obstacle map """
        # 1. get boundaries' values of the environment
        self.min_x = round(min(ox))
        self.max_x = round(max(ox))
        self.min_y = round(min(oy))
        self.max_y = round(max(oy))

        # 2. calculate needed numbers of x,y in map
        self.x_grid_num = round(self.max_x - self.min_x)
        self.y_grid_num = round(self.max_y - self.min_y)

        # 3. obstacle map generation
        self.obstacle_map = [[False for _ in range(self.x_grid_num)] for _ in
 range(self.y_grid_num)]

    def planning(self, sx, sy, gx, gy,CF_N,CF_E,CF_S,CF_W):
        """ dijkstra path search """

        # 1. get start_node, goal_node
        sx_index = self.calc_xy_index(sx, self.min_x)
        sy_index = self.calc_xy_index(sy, self.min_y)
        gx_index = self.calc_xy_index(gx, self.min_x)
```

```
        gy_index = self.calc_xy_index(gy, self.min_y)
        start_node = self.Node(sx_index, sy_index, 0.0, -1)
        goal_node = self.Node(gx_index, gy_index, 0.0, -1)

        # 2. initialize open_set, close_set, put start_node into open_set
        open_set, close_set = dict(), dict()
        open_set[self.calc_index(start_node)] = start_node

        # 3. search
        while True:
            # (1). choose the node whose cost is minimum in open_set
            c_id = min(open_set, key=lambda o: open_set[o].cost)
            current = open_set[c_id]

            if show:
                plt.plot(self.calc_position(current.x, self.min_x),
                         self.calc_position(current.y, self.min_y), "xc")
                # for stopping simulation with the esc key.
                # plt.gcf().canvas.mpl_connect(
                #     'key_release_event',
                #     lambda event: [exit(0) if event.key == 'escape' else None])
                if len(close_set.keys()) % 10 == 0:
                    plt.pause(0.001)

            # (2). determine whether the current node is the end point
            if current.x == goal_node.x and current.y == goal_node.y:
                goal_node.parent_index = current.parent_index
                goal_node.cost = current.cost
                break

            # (3). remove the current node from the open set, add it to the closed set
            del open_set[c_id]
            close_set[c_id] = current

            # (4). expand search grid based on motion model
            self.robot_motion = self.get_motion_model(current.x,current.y,CF_N,CF_E,CF_S,CF_W)
            for move_x, move_y, move_cost in self.robot_motion:
                node = self.Node(current.x + move_x,
                                 current.y + move_y,
                                 current.cost + move_cost, c_id)
                n_id = self.calc_index(node)

                if n_id in close_set:
                    continue

                if not self.verify_node(node):
                    continue

                if n_id not in open_set:
                    open_set[n_id] = node   # discover a new node
                else:
                    if open_set[n_id].cost >= node.cost:
                        # This path is the best until now. record it!
                        open_set[n_id] = node

        rx, ry = self.calc_final_path(goal_node, close_set)

        return rx, ry, goal_node.cost

    def calc_final_path(self, goal_node, close_set):
```

```
        """ generate final course """
        rx = [self.calc_position(goal_node.x, self.min_x)]
        ry = [self.calc_position(goal_node.y, self.min_y)]

        parent_index = goal_node.parent_index
        while parent_index != -1:
            n = close_set[parent_index]
            rx.append(self.calc_position(n.x, self.min_x))
            ry.append(self.calc_position(n.y, self.min_y))
            parent_index = n.parent_index

        return rx, ry

    class Node:
        def __init__(self, x, y, cost,parent_index):
            self.x = x
            self.y = y
            self.cost = cost       # g(n)
            self.parent_index = parent_index
        #
        # def __str__(self):
        #     return str(self.x) + "," + str(self.y) + "," + str(self.cost) + ","
   + str(self.parent_index)
    """
    These 3 functions calc_index, calc_xy_index and calc_position are used for
     coordinate system transformation
    However, in this project, they are useless because values of x and y in grid
     map coordinates and xy map are the same.
    """
    def calc_index(self, node):
        index = node.y * self.x_grid_num + node.x
        return index

    def calc_xy_index(self, pos, min_p):
        index = round(pos - min_p)
        return index

    def calc_position(self, index, min_p):
        pos = min_p + index
        return pos

    def verify_node(self, node):
        """ check whether the current position is appropriate """
        px = self.calc_position(node.x, self.min_x)
        py = self.calc_position(node.y, self.min_y)

        if px < self.min_x or px > self.max_x:
            return False
        if py < self.min_x or py > self.max_y:
            return False

        return True

    @staticmethod
    def get_motion_model(x,y,CF_N,CF_E,CF_S,CF_W):
        # dx, dy, cost
        data_x=x
        data_y=y
        model = [
            [0, 1, CF_N[data_x,data_y+1]],         # North
            [0, -1,CF_S[data_x,data_y-1]],          # South
            [-1, 0,CF_E[data_x-1,data_y]],          # East
```

```
            [1, 0, CF_W[data_x+1,data_y]],        # West
        ]
        return model
# ## for tasks
# ## t,sx,sy,gx,gy,xcol,yrow,num_iter,dist
"""
test 1: time=0.1,num_iter=30     succeed
test 2: time=1,num_iter=30        succeed
test 3: time=10, num_iter=30     succeed
test 4: time=100, num_iter=30   fail
test 5: time=100,num_iter=50    succeed
"""
time=100                            #Given Time [h]
sx, sy = 3, 3                        #x,y of Start Point
gx, gy = 47, 47                   #x,y of Goal Point
xcol=50                             #Number of Columns
yrow=50                            #Number of Rows
num_iter=50                       #Limitation of iterations
# time=float(input('given time(h):'))
t=time*3600          #Given time[s]
dist=1000             #Distance between neighbour nodes[m]
# sx=int(input('start point[x](km):'))
# sy=int(input('start point[y](km):'))
# gx=int(input('goal point[x](km):'))
# gy=int(input('goal point[y](km):'))
# # it is a better choice to choose xcol=yrow
# xcol=int(input('number of columns(>abs(sy-gy)):'))
# yrow=int(input('number of rows(>abs(sx-gx):'))
# num_iter=float(input('Limitation of iterations(max=xcol*yrow):'))
# dist=float(input('Distance between neighbour nodes(m):'))
# ## for wind resistance
# ## Vs,rho_air,Cair,filepath1,filepath2
Axv=1600
rho_air=1.293
filepath1 = "E:/User/Desktop/datalab/u-wind1.csv"
filepath2 = "E:/User/Desktop/datalab/v-wind1.csv"
# Axv=float(input('Area of maximum transverse section exposed to the wind(m^2):')
    )
# rho_air=float(input('Density of Air (kg/m^3):'))
# Cair=np.mat(float(input('wind resistance coefficient(Be careful with
    Cair_extend):')))
# filepath1=input('filepath of wind_u:')
# filepath2=input('filepath of wind_v:')
# ## for total resistance
# ## Vs,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,rho_sea,nu_sea,g
L=200
Lwl=195
Los=196.5
B=33
TF=11.6
TA=11.4
CB=0.855
S=10500
Dp=7
NRud=1
NBrac=1
NBoss=1
NThr=1

rho_sea = 1025
nu_sea = 1.1395E-6
g = 9.81
```

```
# L=float(input('Length of Ship(m):'))
# Lwl=float(input('Length of Water Line(m):'))
# Los=float(input('Length over Surface(m):'))
# B=float(input('Beam(m):'))
# TF=float(input('Draft of Fore Propeller(m):'))
# TA=float(input('Draft of Aft Propeller(m):'))
# CB=float(input('Block coefficient:'))
# S=float(input('Wetted Surface(m^2):'))
# Dp=float(input('Propeller diameter(m):'))
# NRud=float(input('Number of rudders:'))
# NBrac=float(input('Number of brackets:'))
# NBoss=float(input('Number of bossings:'))
# NThr=float(input('Number of side thrusters:'))
# rho_sea=float(input('Density of Sea Water (kg/m^3):'))
# nu_sea=float(input('Viscosity of Sea water (m^2/s):'))
# g=float(input('Gravitational Constant (m/s^2):'))
# ## for added resistance
# ## Vs,Lbwl,B,rho_sea,g,filepath3,filepath4
Lbwl=38
filepath3 = "E:/User/Desktop/datalab/wave_mwd.csv"
filepath4 = "E:/User/Desktop/datalab/wave_swh.csv"
# Lbwl=float(input('Length of the Bow on the Water Line to 95% of maximum Beam (m):')"
# filepath3=input('filepath of wave_d:')
# filepath4=input('filepath of wave_h:')

## Wind Resistance
wind_u = ReadCsvWind(filepath1,xcol,yrow)
wind_v = ReadCsvWind(filepath2,xcol,yrow)

"""
If a new Cair is used, be careful with below functions for Cair_extend
Inital data about General Cargo form ITTC (Value range: 0-180 degrees):
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]
Step 1: Wind resistance coefficient: Data about General Cargo form ITTC, but no data for 130 degree angle
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])
Step 2: Extend this matrix (Value range: 0-360 degrees):
"""
# Step 1
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
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])
# Step 2
Cair_extend=np.mat(np.zeros((Cair.size*2-1)))        #Value Range of Inital Data:0-180 degrees
Cair_extend[0,Cair.size-1]=Cair[0,Cair.size-1]          #Value Range of Extended Data:0-360 degrees
for i in range(Cair.size-1):
    Cair_extend[0,i]=-Cair[0,i]
    Cair_extend[0,2*Cair.size-i-2]=-Cair[0,i]

## Viscous/Friction+ Wave Resistance

## Added Resistance
wave_d = ReadCsvWave(filepath3,xcol,yrow)
wave_h = ReadCsvWave(filepath4,xcol,yrow)

""" Build boundary """
```

```
ox, oy = [], []
for i in range(0, xcol):        #north boundary
    ox.append(i)
    oy.append(xcol)
for i in range(0, yrow):         #east boundary
    ox.append(0)
    oy.append(i)
for i in range(0, xcol):        #south boundary
    ox.append(i)
    oy.append(0)
for i in range(0, yrow):         #west boundary
    ox.append(yrow)
    oy.append(i)
dijkstra = Dijkstra(ox, oy)

T_cost=np.mat(np.zeros((1,num_iter)))
TotalCost=T_cost
show = False
for i in range(num_iter):
    length=(abs(sx-gx)+abs(sy-gy))+i
    """
    find minimum cost in each velocity
    Vs_min=(abs(sx-gx)+abs(sy-gy))/t
    Vs_max=xcol*yrow/t
    """
    Vs=float(length*dist/t)
    delta_T=dist/Vs

    """calculate cost for each given velocity"""
    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)
    CF_m=TotalResCosFun(Vs,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,rho_sea,nu_sea,g,delta_T)
    CF_Wave_N,CF_Wave_E,CF_Wave_S,CF_Wave_W=WaveResCosFun(Vs,Lbwl,B,rho_sea,wave_d,wave_h,delta_T)
    CF_N=CF_Wind_N+CF_Wave_N+CF_m
    CF_E=CF_Wind_E+CF_Wave_E+CF_m
    CF_S=CF_Wind_S+CF_Wave_S+CF_m
    CF_W=CF_Wind_W+CF_Wave_W+CF_m

    """find an efficient path for each given velocity"""
    rx, ry, T_cost[0,i] = dijkstra.planning(sx, sy, gx, gy,CF_N,CF_E,CF_S,CF_W)

    """check task time"""
    if len(rx)*delta_T>t:
        # print('The ship cannot reach the goal point in given time with this speed',Vs ,' [m/s]')
        print('The ship cannot reach the goal point in given time with this speed',Vs/0.5144 ,' [knot]')
        TotalCost[0,i]=0
    else:
        # print('The ship could reach the goal point in given time with this speed',Vs ,' [m/s]')
        print('The ship could reach the goal point in given time with this speed',Vs/0.5144 ,' [knot]')
        TotalCost[0,i]=T_cost[0,i]
if np.max(TotalCost)==0:
    print('The effective path could not be found after ',num_iter,' iterations' )
else:

    """compare total costs of paths"""
    minx,miny= np.where(TotalCost == np.min(TotalCost[np.nonzero(TotalCost)]))
```

```
    """get optimum velocity"""
    length_E=(abs(sx-gx)+abs(sy-gy))+miny
    Vs_E=float(length_E*dist/t)
    delta_TE=dist/Vs_E
    CFE_Wind_N,CFE_Wind_E,CFE_Wind_S,CFE_Wind_W = WindResCosFun(Vs_E,Axv,rho_air,wind_u,wind_v,Cair_extend,delta_T)
    CFE_m=TotalResCosFun(Vs_E,L,Lwl,Los,B,TF,TA,CB,S,Dp,NRud,NBrac,NBoss,NThr,rho_sea,nu_sea,g,delta_T)
    CFE_Wave_N,CFE_Wave_E,CFE_Wave_S,CFE_Wave_W=WaveResCosFun(Vs_E,Lbwl,B,rho_sea,wave_d,wave_h,delta_T)
    CFE_N=CFE_Wind_N+CFE_Wave_N+CFE_m
    CFE_E=CFE_Wind_E+CFE_Wave_E+CFE_m
    CFE_S=CFE_Wind_S+CFE_Wave_S+CFE_m
    CFE_W=CFE_Wind_W+CFE_Wave_W+CFE_m
    show = True
    if show:
        plt.plot(ox, oy, '.k')
        plt.plot(sx, sy, 'og')
        plt.plot(gx, gy, 'or')
        # plt.grid('True')
        plt.axis('equal')
        # plt.show()

    """get optimum path for this optimum velocity"""
    rx, ry, TotalCost_min = dijkstra.planning(sx, sy, gx, gy,CFE_N,CFE_E,CFE_S,CFE_W)
    if show:
        print('Find the effective path' )
        # print('The efficient velocity is [m/s]:',Vs_E)
        print('The efficient velocity is [knot]:',Vs_E/0.5144)
        # print('The minimum cost is [J]:',TotalCost_min)
        print('The minimum cost is [MJ]:',TotalCost_min/1000000)
        plt.plot(rx, ry, '-r')
        plt.pause(0.01)
        plt.show()
```

Listing A.10: Testing codes (section 4.2)