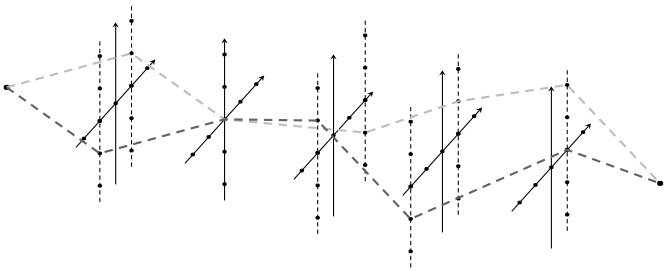
CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Lund University Publications - Student Papers

Multi-Objective Optimization of Voyage Plans for Ships



Waqar Hameed

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
ISRN LUTFD2/TFRT--5994--SE
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2015 by Waqar Hameed. All rights reserved.
Printed in Sweden by Tryckeriet i E-huset
Lund 2015

Abstract

In this thesis two methods are investigated to solve a multi-objective optimization problem for voyage planning. The first method, grid search, is a brute force search in a three-dimensional graph while the other uses the Lipschitzian algorithm DIRECT to do a continuous search along a nominal route.

The grid search method gives a computation time of 7.6 minutes for a route from Gothenburg to New York. This is obtained partly by parallelizing on 10 cores but also implementing core routines efficiently in compiled programming languages.

However, the continuous search method with DIRECT is not suitable for a realistic voyage planning problem. It is more due to the nature of the DIRECT algorithm than the implementation details.

Keywords: multi-objective optimization, voyage plan, Pareto sample, direction method, grid search, Lipschitzian optimization, DIRECT

Acknowledgments

First, I would like to express my sincere gratitude to my supervisor, Mats Molander, for his guidance and support. I appreciate the fact that he always had time to answer my questions, moreover, with such great enthusiasm.

My sincere thanks also goes to Pontus Giselsson and Kateryna Mishchenko, for their valuable suggestions. Further, a special thanks to Marcus Johansson for assisting me with the computer cluster.

This master's thesis was done in conjunction with ABB Corporate Research in Västerås, Sweden. A huge thanks to Karl-Erik Årzén, Rickard Lindkvist and Shiva Sander-Tavallaey for the opportunity.

Last but definitely not least, I would like to express my deepest gratitude to my family and friends. It would most certainly not have been possible to graduate without you.

Contents

1 | Introduction

Optimal voyage planning for an ocean going ship is quite a complex problem. In most existing voyage planning tools, only a single objective is considered for optimization, most commonly the fuel consumption. However, in reality, there are several possibly conflicting criteria that should preferably be optimized.

In addition to minimizing the fuel consumption, some examples are minimizing the travel time, maximizing safety and comfort. This leads to an optimization problem comprising more than one objective function to be optimized together. In order for the voyage planning tool to be practical usable, it should be computationally efficient and at the same time yield credible results.

1.1 Objective

The aim of this thesis is to analyze and develop a computationally efficient voyage planning tool for ships based on several objectives. For this, several different methods will be studied and applied to realistic test problems.

1.2 Outline

The report is structured such that the necessary background theory is documented first: Chapter 2 is a brief introduction to multi-objective optimization and a description of the theory used in the methods. Chapter 3 presents a Lipschitzian algorithm, also used in the methods. In chapter 5 we describe in detail how the voyage planning is modeled.

Thereafter, we document the methods and report the results: Chapter 4 presents some previous work relevant for this study. The main methods of the study is described in Chapter 6 and the results can be found in chapter 7.

Finally, the report concludes with a discussion and conclusions in Chapter 8.

2 | Multi-objective optimization

"Do we actually minimize our sorrows when trying to maximize our happiness?"

Multi-objective optimization is exactly what it sounds like, optimization problems with several objective functions. However, there are some important differences from "ordinary" single-objective optimization. A brief introduction to the theory of multi-objective optimization is presented in this chapter. (Most of the notions and definitions in this chapter comes from [12].)

2.1 The problem

A multi-objective optimization problem can be stated as the following

$$\begin{aligned} &\underset{\boldsymbol{x}}{\text{Minimize}} && \boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})) \\ &\text{subject to} && \boldsymbol{x} \in S \end{aligned} \tag{2.1}$$

where $\boldsymbol{f}$ is a vector function defined by m[1] objective functions $f_i : \mathbb{R}^n \to \mathbb{R}$, $(i = 1, \ldots, m)$, and the subspace $S \subseteq \mathbb{R}^n$ is the so called *feasible region*, which is usually formed from a set of constraint functions.

The vector $\boldsymbol{x}$, to minimize over, is called the *decision vector* and belongs to the *decision space* $\mathbb{R}^n$. The space $\mathbb{R}^m$, which $\boldsymbol{f}$ maps to, is called the *objective space*. The image of the feasible region, that is $\boldsymbol{f}(S) \subseteq \mathbb{R}^m$, is called the *feasible objective space* and is denoted by Z. We denote points in this subspace, the *objective vectors*, with $\boldsymbol{z}$. See Figure 2.1 for an illustration.

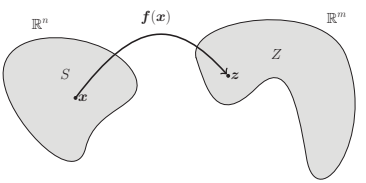


Figure 2.1: The vector function $\boldsymbol{f}$ mapping a decision vector $\boldsymbol{x}$ to an objective vector $\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{x})$.

[1] We assume that $m \geq 2$, otherwise we have an ordinary single-objective optimization problem.

Now, what do we exactly mean by minimizing $\boldsymbol{f}$ in (2.1)? Since $\boldsymbol{f}$ is a vector function, "minimize" here means to minimize the objective functions f_i all together. If some objective function f_i instead needs to be maximized, we can easily reformulate that to an equivalent minimizing problem by setting $-f_i$. Thus, we will always refer to a minimizing problem as in (2.1).

Moreover, in order for this to be nontrivial, we require that the objective functions are conflicting. That is, improving one objective function f_i worsens some other f_j. For example, minimizing the time for a voyage and minimizing the fuel consumption of the vehicle are conflicting objectives (since a shorter trip time requires more power, which in turn yields higher fuel consumption). Therefore, there can not exist a single solution that is optimal in regard to every objective function.

2.2 Pareto optimality

As we have seen, a multi-objective optimization problem with conflicting objective functions does not have a single global optimal solution. The concept of optimality in this case comes from the following definition.

Definition 2.1. A point $\boldsymbol{z}^* = (z_1^*, \ldots, z_m^*) \in Z \subseteq \mathbb{R}^m$, is said to be *Pareto optimal* if and only if there does not exist another point, $\boldsymbol{z} = (z_1, \ldots, z_m) \in Z$, such that $\boldsymbol{z} \leq \boldsymbol{z}^*$ and $z_i < z_i^*$ for at least some $i = 1, \ldots, m$. (Note that $\boldsymbol{z} \leq \boldsymbol{z}^*$ is a vector inequality.)

In other words, an objective vector $\boldsymbol{z}^*$ is said to be Pareto optimal if there is no way of improving an objective function without deteriorating another. The corresponding point to $\boldsymbol{z}^*$ in the decision space, $\boldsymbol{x}^*$, is then also said to be Pareto optimal.

The set of all the Pareto optimal solutions is called the *Pareto front*. Figure 2.2 shows some examples of Pareto fronts in $\mathbb{R}^2$. Notice that the Pareto front does not need to be connected.

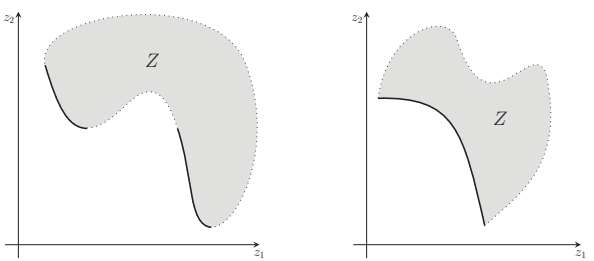


Figure 2.2: Two examples of Pareto fronts (highlighted with thick lines) in the objective space.

2.3 Ideal and nadir point

If the feasible objective space Z is bounded, then the Pareto front (the set of all Pareto optimal solutions) also have an upper respective lower bound.

Definition 2.2. The coordinates of the *ideal point* $\boldsymbol{z}_I = (z_{I1}, \ldots, z_{Ii}, \ldots, z_{Im}) \in \mathbb{R}^m$ are the solutions to

$$\begin{aligned} &\underset{\boldsymbol{x}}{\text{Minimize}} && f_i(\boldsymbol{x}), \quad \text{for } i = 1, \ldots, m. \\ &\text{subject to} && \boldsymbol{x} \in S \end{aligned}$$

where the terms are the same as in (2.1). This can also be stated equivalently as $z_{Ii} = \inf_{\boldsymbol{x} \in S} f_i(\boldsymbol{x})$.

The ideal point gives rise to a lower bound to the Pareto front, see Figure 2.3 for an illustration. However, this point can never be feasible when dealing with conflicting objective functions. It would otherwise be the global optimal solution, and as we already know by now, there is none with conflicting objectives.

The point responsible for the upper bound is called the *nadir point*.

Definition 2.3. The coordinates of the *nadir point* $\boldsymbol{z}_N = (z_{N1}, \ldots, z_{Ni}, \ldots, z_{Nm}) \in \mathbb{R}^m$ are the solutions to

$$\begin{aligned} &\underset{\boldsymbol{x}}{\text{Maximize}} && f_i(\boldsymbol{x}), \quad \text{for } i = 1, \ldots, m. \\ &\text{subject to} && \boldsymbol{x} \in S \text{ and is Pareto optimal.} \end{aligned}$$

where the terms are the same as in (2.1). This can also be stated equivalently as $z_{Ni} = \sup f_i(\boldsymbol{x})$, where the supremum is taken over all $\boldsymbol{x} \in S$ that is Pareto optimal.

The nadir point is thus the worst value of the objective functions in the Pareto front. Note however that this is not the same as the worst value obtainable in the whole objective space, see Figure 2.3.

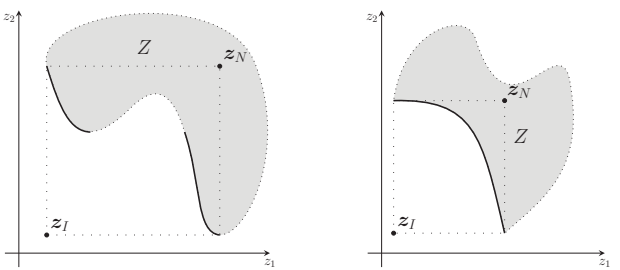


Figure 2.3: Two examples of the ideal and nadir points in $\mathbb{R}^2$ (cf. Figure 2.2).

2.4 Methods classification

Formally, when the Pareto front has been found, the problem (2.1) is considered to be solved. However, we are then left with a (possibly infinite) set of solutions and one might only be interested in obtaining one. This is where a *decision maker* comes into the picture. The decision maker is someone (or maybe even *something*) that chooses a solution according to their preferences.

Most of the solving methods for multi-objective optimization problems can be categorized by how the decision maker comes into play. A method can be classified into one (or even several) of the following categories:

No-preference methods
The decision maker's preferences are simply ignored. A single solution is obtained using some relatively simple method and is just presented to the decision maker (who can either accept or reject it).

A priori methods
The decision maker states the preferences beforehand. The method then takes this in to consideration when calculating a solution.

A posteriori methods
The Pareto front (or a sample of it) is calculated from an already predefined method. It is then up to the decision maker to choose one preferred solution.

Interactive methods
The decision maker is active through the whole solution process. These methods iteratively asks the decision makers preference and updates the solutions accordingly, until a final solution is obtained.

We will later use an a posteriori method. The reason an a posteriori method is chosen over an a priori one is the following: It is usually very hard to achieve a solution sufficiently close to some already defined parameters. Moreover, since the Pareto front is initially unknown, it is difficult for the decision maker to know beforehand which solutions are actually obtainable. Most of the times the solution would therefore nevertheless dissatisfy the decision maker, meaning that the method has to run several times before obtaining an acceptable solution. This would imply, in the long run, a more computationally intensive method.

2.5 Obtaining the Pareto front

Suppose we have a finite set Z of points in the feasible objective space, see Figure 2.4. How do we find the Pareto front? Below we present some different methods that we will use later for obtaining the Pareto front.

2.5.1 Scaling

To make the calculations easier, we can scale each point $\boldsymbol{z} = (z_1, \ldots, z_n)$ in Z as

$$z'_i = \frac{z_i - z_{Ii}}{z_{Ni} - z_{Ii}}, \quad \text{for } i = 1, \ldots, n, \tag{2.2}$$

where z_{Ii} and z_{Ni} are the coordinates of the ideal and nadir point, respectively [12]. This makes sure that the Pareto front of the new scaled points $\boldsymbol{z}' = (z'_1, \ldots, z'_n)$ falls in the range $[0, 1]$.

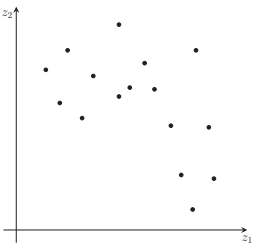


Figure 2.4: A set of points in the feasible objective space.

2.5.2 Filtering and sampling

By using definition 2.1, it can easily be checked if a point belongs to the Pareto front or not. One algorithm for obtaining the Pareto front is thus simply to check the comparison criteria in the definition for every other point. Figure 2.5 shows an example of this Pareto filtering algorithm applied to the set of points in Figure 2.4.

The efficiency of the algorithm can be further reduced if one also mark already checked dominated points[2]. This means that in future comparisons the dominated points will not then be considered, resulting in fewer comparisons.

One might also want to obtain just a portion of the Pareto front, say only n numbers of points. One way to obtain a set of points spread out as much as possible is to choose them so that the minimum distance between any two points is maximized. For instance, in Figure 2.6 we sample the Pareto front in Figure 2.5 with $n = 3$.

[2]We say that a point $\boldsymbol{x}$ is dominated if there exist another point $\boldsymbol{x}^*$ which is better with respect to all objectives, i.e. $\boldsymbol{f}(\boldsymbol{x}^*) < \boldsymbol{f}(\boldsymbol{x})$.

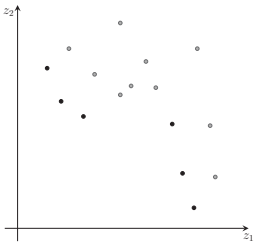


Figure 2.5: Pareto filtering. The dominated points are highlighted in gray.

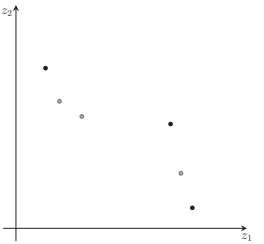


Figure 2.6: Pareto sampling. The sampled points are highlighted in black.

2.5.3 Direction method

Another algorithm that we will later use is a slight modification of the so called direction method [14]. This methods *scalarizes* the problem (2.1) into a set of single-objective optimization problems. Each single-objective solution then gives rise to one point on the Pareto front.

Suppose that we have scaled the points in the objective space $\mathbb{R}^m$ as in (2.2). The Pareto front will thus be in the unit hypercube. Now let us project the vertices of the hypercube onto the plane outside the hypercube with the normal $(1, \ldots, 1)$, we will call this plane the *reference plane*, see Figure 2.7 for the situation in $\mathbb{R}^3$.

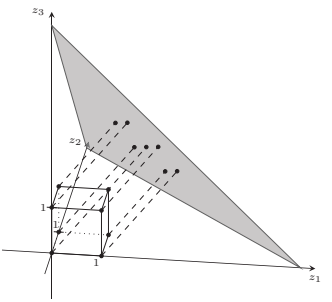


Figure 2.7: Projecting the unit cube's vertices onto a plane with the normal $(1, 1, 1)$. Note that the vertices $(0, 0, 0)$ and $(1, 1, 1)$ have the same projection on the plane.

Next, we determine the convex hull of the projected points in the reference plane. Some of the points will be in the interior, see Figure 2.8.

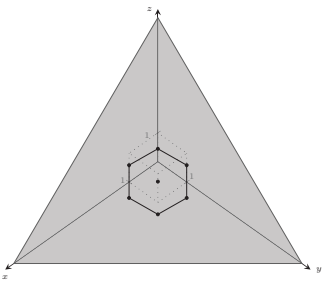


Figure 2.8: The convex hull of the projected points on the plane. The unit cube is dotted and is in this figure behind the plane.

The idea now is to beam *search lines* from *reference points* inside this convex hull downwards, in the direction $(-1, \ldots, -1)$, into the hypercube (search lines outside this convex hull will obviously "miss" the hypercube). Using convex cones along the lines we can find and sample the Pareto front.

More specifically, consider Figure 2.9. From a reference point $\boldsymbol{z}_0$ on the reference plane (which in $\mathbb{R}^2$ becomes a line), a search line beams in the direction $\boldsymbol{d}$ into the set of points. Along the points on this line, i.e. $\boldsymbol{z}_0 + t\boldsymbol{d}$ where $t \in \mathbb{R}$, we subtract the cone $\mathbb{R}^m_+$. If there is only one point, say $\boldsymbol{z}^*$, in this resulting set then it must be Pareto optimal, that is

$$(\boldsymbol{z}_0 + t\boldsymbol{d} - \mathbb{R}^m_+) \cap Z = \boldsymbol{z}^* \implies \boldsymbol{z}^* \text{ is Pareto optimal.} \tag{2.3}$$

This is because by definition, $\boldsymbol{z}^*$ is then Pareto optimal (see Definition 2.1).

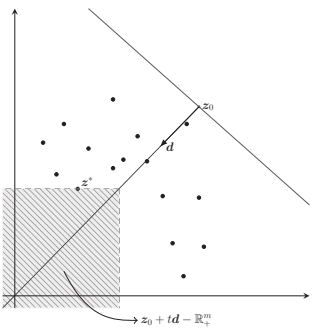


Figure 2.9: A search line from the reference plane beaming into a set of points in the feasible objective space. The point $\boldsymbol{z}^*$ is Pareto optimal.

To find $\boldsymbol{z}^*$ for one particular search line, we calculate the t-value for each point $\boldsymbol{z}$ in Z such that $\boldsymbol{z}$ is on the boundary of the set $\boldsymbol{z}_0 + t\boldsymbol{d} - \mathbb{R}^m_+$, as in Figure 2.9. The point with the largest t-value will then fulfill (2.3), and is the sought Pareto optimal point $\boldsymbol{z}^*$.

How do we calculate the t-value for one particular point $\boldsymbol{z}$? The condition such that $\boldsymbol{z}$ is on the boundary of the set $\boldsymbol{z}_0 + t\boldsymbol{d} - \mathbb{R}^m_+$ is

$$\boldsymbol{z}_0 + t\boldsymbol{d} \geq \boldsymbol{z} \iff z_{0i} + td_i \geq z_i, \text{ for } i = 1, \ldots, m,$$

with equality for at least one i. This yields (recall that $\boldsymbol{d} < 0$)

$$t \leq \frac{z_i - z_{0i}}{d_i}, \text{ for } i = 1, \ldots, m. \implies t = \min_i \frac{z_i - z_{0i}}{d_i}. \tag{2.4}$$

Figure 2.10 shows the resulting obtained Pareto front when applying the direction method on the points in Figure 2.4, with only three reference points.

Remark. When creating reference points on the reference plane in the convex hull, we obviously want them to be equidistant. This is because the sampling of the Pareto front will then also be somewhat equidistant.

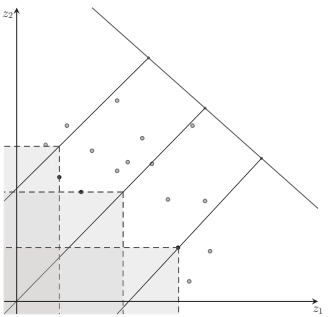


Figure 2.10: The direction method with three reference points. The sampled points are highlighted in black.

2.5.4 Weighting method

We end this chapter by describing one frequently used scalarizing method, the *weighting method*. In the weighting method, each objective function is assigned a weight. We then turn the problem into a single-objective optimization by minimizing the weighted sum

of these. More specifically (with the terms as usual from (2.1))

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{Minimize}} \quad & \sum_{i=1}^{m} \lambda_i f_i(\boldsymbol{x}) \\ \text{subject to} \quad & \boldsymbol{x} \in S \\ & \lambda_i \geq 0, \quad \text{for } i = 1, \ldots, m \\ & \sum_{i=1}^{m} \lambda_i = 1 \end{aligned} \tag{2.5}$$

It might not be immediately clear if the solution to (2.5) really gives a Pareto optimal solution. The following theorem tells us however that this really is the case.

Theorem 2.1. The solution to (2.5) is Pareto optimal if the weights are strictly positive, that is $\lambda_i > 0$ for all $i = 1, \ldots, m$.

Proof. We prove the theorem by contradiction.

Let $\boldsymbol{x}^* \in S$ be a solution to (2.5), with strictly positive weights. Now suppose that $\boldsymbol{x}^*$ is not Pareto optimal. From definition 2.1, this means that there exists a solution $\boldsymbol{x} \in S$ such that

$$\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{x}) \leq \boldsymbol{z}^* = \boldsymbol{f}(\boldsymbol{x}^*)$$

and

$$z_i = f_i(\boldsymbol{x}) < z_i^* = f_i(\boldsymbol{x}^*)$$

for at least some $i = 1, \ldots, m$.

But since all $\lambda_i > 0$, we then have

$$\sum_{i=1}^{m} \lambda_i f_i(\boldsymbol{x}) < \sum_{i=1}^{m} \lambda_i f_i(\boldsymbol{x}^*)$$

which is a contradiction to our initial assumption that $\boldsymbol{x}^*$ is a solution to (2.5). Thus, $\boldsymbol{x}^*$ is Pareto optimal. □

Remark. The assumption, all λ_i strictly positive, in the theorem can be relaxed. However, the solution will then only be so called *weakly Pareto optimal*. We will not go into any further details.

A drawback of the weighting method is that, if the Pareto front is non-convex, it can not find all the points on the Pareto front. We will conclude this chapter with a heuristic example of this below.

In two dimensions, the scalar objective function to minimize in (2.5) has the form

$$\lambda_1 f_1(\boldsymbol{x}) + \lambda_2 f_2(\boldsymbol{x}) = \lambda_1 z_1 + \lambda_2 z_2$$

which, after substituting the constraint $\lambda_1 + \lambda_2 = 1$, simplifies to

$$\lambda_1 z_1 + (1 - \lambda_1) z_2.$$

This function's level curves are lines

$$D = \lambda z_1 + (1 - \lambda) z_2 \iff z_2 = -\frac{\lambda}{1 - \lambda} z_1 + \frac{D}{1 - \lambda}$$

where $D \in \mathbb{R}$ is a constant (for simplicity we dropped the index for λ_1). Now, when varying the weight λ in the interval $[0, 1]$, we see that the slope of the line varies in the interval $]-\infty, 0]$. However, no matter what value λ takes, there is no way to obtain a solution for (2.5) in the non-convex region. Figure 2.11, Figure 2.12 and Figure 2.13 depicts this situation.

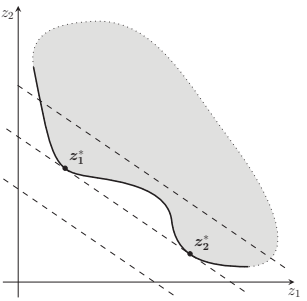


Figure 2.11: A non-convex Pareto front with level curves (dashed lines) from the weighting method. Here two points are found just outside the non-convex region (between $\boldsymbol{z}_1^*$ and $\boldsymbol{z}_2^*$).

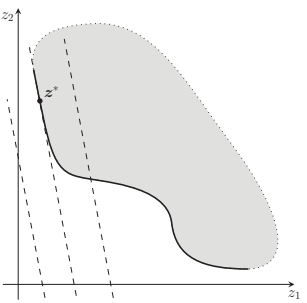


Figure 2.12: A greater λ value gives more steeper level curves. In this case, a solution is found to the left of the non-convex region.

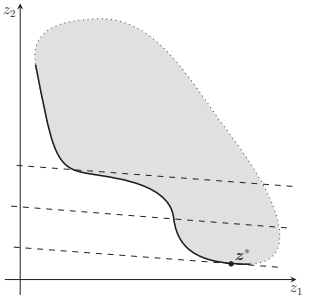


Figure 2.13: A smaller λ value gives more gentle level curves. Here the solution is found to the right of the non-convex region.

Also, it might be very difficult to choose exactly which points that one would like to obtain on the Pareto front by only changing the weights λ_i. Figure 2.14 and Figure 2.15 shows some examples of this.

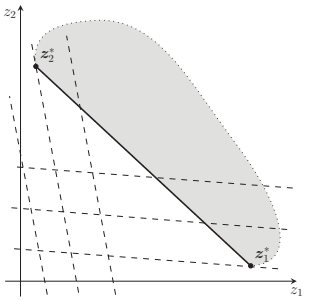


Figure 2.14: Only for a specific value for the weights will the whole Pareto front be obtainable. For other weight values, only the end points $\boldsymbol{z}_1^*$ and $\boldsymbol{z}_2^*$ will be obtained.

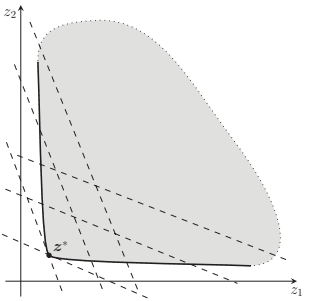


Figure 2.15: Only for a some specific values for the weights will other points other than $\boldsymbol{z}^*$ on the Pareto front be obtainable.

3 | DIRECT algorithm

Lipschitzian optimization algorithms exploit a key property of the objective function to be minimized: How fast the function can "change", also known as the Lipschitz constant. However, in most cases one might not know much about this property. In this chapter we present one algorithm, known as DIRECT, that claims to solve this issue. (This chapter is based on the original paper [10].)

The DIRECT (DIviding RECTangle) algorithm tries to solve some problems with the so called Shubert's algorithm [15]: First, one has to know the Lipschitz constant. Even an estimate of K could result in poor convergence (e.g. too large K). Second, current implementations in higher dimensions is not computational efficient. For instance, in the one-dimensional case the algorithm initializes by evaluating the function at the endpoints a and b. However, in n dimensions, this means evaluating 2^n vertices of a hyperrectangle.

We will first begin by showing DIRECT in the one-dimensional case, then present the full algorithm in the multidimensional case.

3.0.5 One-dimensional case

Consider a one-dimensional function $f(x)$ on a closed interval $[a, b] \subset \mathbb{R}$. Furthermore, assume that it is Lipschitz continuous, that is there exist a $K > 0$ such that

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2| \tag{3.1}$$

for all x_1, $x_2 \in [a, b]$. K is called the *Lipschitz constant*. This means that the absolute value of the derivative of f is always less than K, and thus a constraint of how fast f can "change". Let us sample the center point $c = \frac{a+b}{2}$. Using (3.1), one can determine a lower bound for f:

$$\begin{aligned} f(x) &\geq f(c) + K(x - c), \quad \text{for } x \leq c, \\ f(x) &\geq f(c) - K(x - c), \quad \text{for } x \geq c. \end{aligned}$$

This corresponds to two lines with slopes $-K$ and K through the point $f(c)$ bounding f from below, see Figure 3.1. From Figure 3.1 we see that the lowest value f can obtain in the interval occurs at the endpoints, with the value

$$B_c(a, b, f, K) = f(c) + K(a - c) = f(c) - \frac{K}{2}(b - a). \tag{3.2}$$

Notice how we obtained a lower bound with only one function evaluation at the center point.

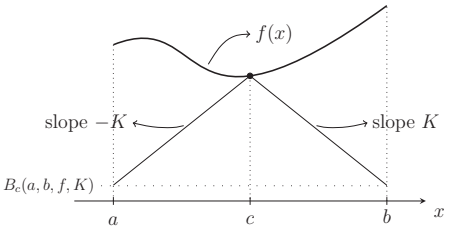


Figure 3.1: Lipschitz continuous function bounded from below using center-point sampling.

Next, we need to divide the interval and choose a new subinterval with lower B_c-value. Since we want to sample the center points in any interval, the subdividing of the intervals will be in thirds, see Figure 3.2. The new center points will be the centers of these smaller intervals, with each having a length of a third of the original interval.

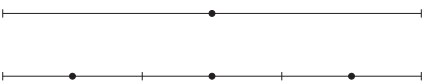


Figure 3.2: Subdividing an interval (top) into thirds and sampling new centers (bottom).

Now, how should we choose the new intervals with lower B_c-values? Assume that we have already partitioned into m subintervals $[a_i, b_i]$, $i = 1, \ldots, m$. In Figure 3.3, each of these intervals is represented by a dot. The horizontal coordinate is the distance from the center point to an endpoint, while the vertical coordinate is the function value at the center point. If we now draw a line with the slope K through any of these points, the intersection with the vertical axis will be the corresponding interval's B_c-value, cf. (3.2).

The idea now is to pick the intervals with the lowest B_c-value for all possible values for the Lipschitz constant K, i.e. the slope of the line. These *potentially optimal* intervals are picked in Figure 3.4 (we will state the formal definition of "potentially optimal" in the next section). As one quickly realizes, the potentially optimal intervals correspond to the lower right convex hull of the points in Figure 3.3.

Thus, we sample all the potentially optimal intervals in one iteration. The algorithm we will stop after some specified number of iterations or function evaluations.

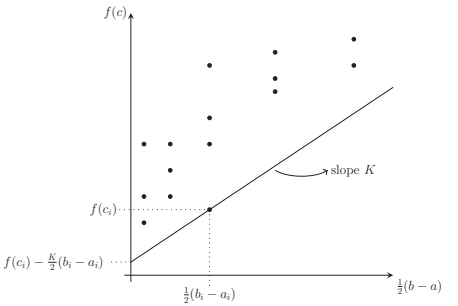


Figure 3.3: Graphical representation of intervals.

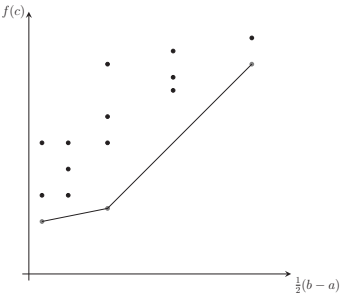


Figure 3.4: Potentially optimal intervals (highlighted as gray points).

3.0.6 Multidimensional case

It is pretty straightforward to extend the procedure in the previous section to several dimensions. The only thing that might cause some trouble is how to divide the hyperrectangles so that each sampling point remains as a center point.

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function of several variables. Without loss of generality, assume that the domain of f is inside the n-dimensional unit hypercube, which can always be achieved by scaling. Let us start by sampling the points $\boldsymbol{c} \pm \delta \boldsymbol{e}_i$, $i = 1, \ldots, n$, where $\boldsymbol{c}$ denotes the center point in the hypercube, δ is one third of the hypercube's side length and $\boldsymbol{e_i}$ is the unit vector along the dimension i.

Figure 3.5 depicts the situation in two dimensions. In the figure, sampled points are shown as circles with their function values inside them. The figure also shows two possible ways to divide the unit square. In (a), we first divide into thirds horizontally then divide the middle subrectangle vertically. In (b), we do this instead in the opposite

order. Although both ways make sure that the sampled point becomes the center points in the subrectangles, DIRECT uses the strategy in (b).

More specifically, if $y_i = \min(f(\boldsymbol{c} - \delta \boldsymbol{e}_i), f(\boldsymbol{c} + \delta \boldsymbol{e}_i))$, DIRECT divides a hypersquare along the dimension with the lowest y_i-value. Thereafter, we divide the hyperrectangle containing the center point $\boldsymbol{c}$ along the dimension with the next lowest y-value. We continue in this manner until a division has been made on all dimensions. This is precisely the strategy used in (b).

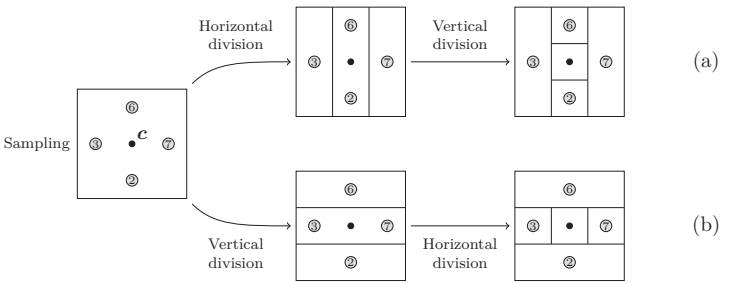


Figure 3.5: Sampling the unit square and two different ways to divide it. The sampled points are represented as gray circles with the corresponding function value inside them. The upper path (a) first divides on the horizontal then on the vertical. The bottom path (b) divides vice versa.

Moreover, notice that we only divide rectangles along the dimension with longest side length. Thus, for hyperrectangles, the same strategy is used as described above for the hypersquare, but the division is only made along the dimensions with the longer sides (starting with the dimension with the lowest y-value and so on).

Now, all that remains is an explanation on how to pick the so called potentially optimal rectangles. As previously promised, here is the definition.

Definition 3.1. Assume that the unit hypercube has been partitioned into m hyperrectangles. Let $\boldsymbol{c}_i$ denote the ith hyperrectangle's center point and let d_i be the distance from this center point to the vertices. Further, let $f_{\min}$ be the current lowest evaluated function value and $\varepsilon > 0$. A hyperrectangle j is said to be *potentially optimal* if there exists some $K > 0$ such that

$$\begin{aligned} &f(\boldsymbol{c}_j) - K d_j \leq f(\boldsymbol{c}_i) - K d_i, \qquad \text{for all } i = 1, \ldots, m, \\ &f(\boldsymbol{c}_j) - K d_j \leq f_{\min} - \varepsilon |f_{\min}|. \end{aligned}$$

The first condition in the definition means that a potentially optimal hyperrectangle has the lowest lower bound for some $K > 0$, cf. (3.2). Thus belonging to the lower right convex hull in Figure 3.4.

The second condition states that this lower bound is smaller than $f_{\min}$ by a nontrivial amount. This is to prevent the algorithm from searching too "local". Nevertheless, DIRECT always converges as long as the function is continuous, or at least in the neighborhood of the global optimum. For the full proof, see [10, p. 173].

We end this chapter with the formal description of the DIRECT algorithm.

Algorithm 3.1 DIRECT

```
1: normalize the function's domain to the unit hypercube
2: c_1 ← center point of the unit hypercube
3: f_min ← f(c_1)
4: m ← 1                                          ▷ number of samples
5: k ← 0                                          ▷ number of iterations
6: while k < maximum number of iterations and
   m < maximum number of function evaluations do
7:     S ← identify the set of potentially optimal rectangles
8:     while S ≠ ∅ do
9:         j ← any rectangle in S
10:        sample and divide rectangle j           ▷ as described above
11:        f_min ← new lower sampled f_min         ▷ update f_min
12:        m ← m + number of new points sampled
13:        S ← S − {j}
14:    end while
15:    k ← k + 1
16: end while
```

4 | Previous work

In this chapter, a handful of some already existing methods for voyage planning relevant for this work will be presented. (A more extensive review can be found in [1].)

4.1 Single-objective

4.1.1 Isochrone method

One of the earliest method in voyage planning, which also takes the weather in consideration, is the so called *isochrone method* [9]. Starting from a point and considering all possible directions, an *isochrone* is a set of points that one can reach within a certain time limit. These points are of course dependent on the weather, e.g. such as wave height and direction. The method is recursive such that after the first isochrone, a second one is calculated starting from the first one and so on, see Figure 4.1. Consequently, the methods main focus, and only, is time. The fastest route is thus the one passing through the fewest isochrones.

Although the method was originally meant to be used manually, computer implementations encountered problems. The main one being so called "isochrone loops". This is an irregularity in the isochrones shape caused by non-convexity [17]. However, further improvements of the method reduced this problem [6].

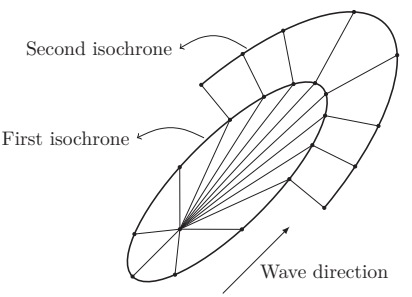


Figure 4.1: Formation of the first and second isochrone.

4.1.2 Dynamic programming

In the context of voyage route optimization the term dynamic programming is used for a method where the best route is searched in a prespecified grid (the term grid search will be used in the rest of the report). The method can be described as follows: Suppose we have a nominal route between the departure point and arrival point. We grid this nominal route with a number of *stages* and each stage containing points extending orthogonally on both side of the route (except the departure and arrival stage), see Figure 4.2. This defines a discrete search space.

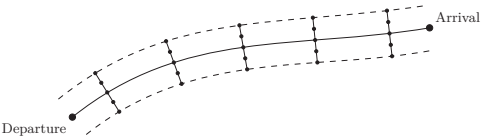


Figure 4.2: A nominal route gridded with 7 stages (5 stages plus the arrival and destination stage) with two points on each side (in total five points per stage).

The idea is to calculate the objective and constraint functions for all possible combinations of nodes in the grid and find the optimal ones. Grid search is thus essentially a graph algorithm. Note that we must visit each stage in the grid and are also not allowed to move "backwards", i.e. visit an already traversed stage.

An early reference is [4], which has inspired the development of a state of the art route planning tool VVOS by the company Jeppesen. The company ABB has developed a prototype tool (implemented in MATLAB) based on the same idea.

The main difference is the way varying speeds are treated. Knowing the upper and lower speed limit of the ship, and the time span for the arrival time, one can calculate the possible time spans that are allowed for a specific node in each stage. We can then extend the grid into a three dimensional grid, see Figure 4.3 where the points in the z-direction are the feasible time points for the specific node.

Now, this is how the traversing in the grid is done: For all possible combinations of the *source* nodes in stage k and *destination* nodes in stage $k+1$, we calculate the objective and constraint functions. The infeasible routes (combinations), i.e. routes that violate any constraints (including infeasible time points), are discarded.

We are now left with a set of routes to each destination node. The best one is then saved, and the rest discarded.

We continue with the next stage in the same manner: With the nodes picked in stage $k+1$ (corresponding to current optimal routes) as the new source nodes and so on.

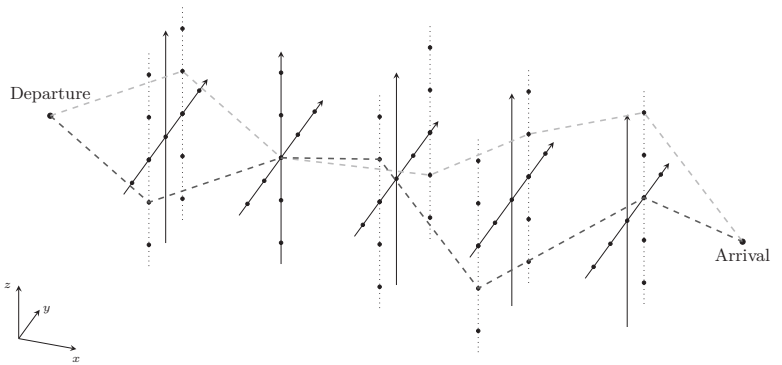


Figure 4.3: Grid search in a three dimensional grid. The x- and y-nodes corresponds to position, while the z-nodes are time points. Two different routes in the grid have been drawn (dashed with different shades).

4.1.3 Optimal control

Optimal control (using calculus of variation) has been used for single-objective voyage planning [7, 3, 2]. The control variables in the optimization problem are the heading and the velocity. These can be chosen freely and can thus theoretically give the optimum with arbitrary accuracy.

However, the method requires differentiable objective and constraint functions. There are also convergence and sensitivity issues relating to the initial conditions (nominal route) [13].

4.2 Multi-objective

4.2.1 Evolutionary algorithms

A common method to solve multi-objective optimization problems is by using evolutionary algorithms and the subclass genetic algorithms. The idea is inspired from the evolution in nature. Concepts such as mutation, natural selection and recombination are used to find an optimal solution. Examples of such algorithms are the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) and the Strength Pareto Evolutionary Algorithm (SPEA) [5, 18].

Evolutionary algorithms have also been applied for multi-objective voyage planning, with the objectives travel time and fuel consumption [7]. However, the main disadvantages

of these algorithms are the long computation time and the uncertainty of the solutions Pareto optimality [1].

4.2.2 Grid search

In [16] a multi-objective version of the grid search algorithm is described where at each node instead of only saving the single-objective optimal routes, all Pareto optimal ones are saved.

In [1] the ideas in [16] are used to include multi-objective optimization functionality into the ABB prototype voyage planning tool. One difference to the method in [16] is that only a number of samples of the Pareto front is saved in each node.

5 | Voyage planning

When applying the theory of multi-objective optimization to voyage planning, the details in the minimization problem (2.1) of course need to be specified. In this chapter we will describe the objective and constraint functions. Also, the different test problems that will be used in the methods will be presented.

5.1 The objective functions

There are in total three objective functions that will be considered. The first one is $f_1(\boldsymbol{x})$ = "total time". That is, with a certain decision vector $\boldsymbol{x}$, f_1 is the total time of the journey (in hours), from departure to arrival (the exact dimension and structure of $\boldsymbol{x}$ will be described in more detail in the next chapter)

The second objective function is $f_2(\boldsymbol{x})$ = "total fuel consumption". That is, f_2 is the total fuel consumed (in tonnes) in the journey. The fuel consumption is calculated according to an already predefined model of the ship in regard: Figure 5.1 and Figure 5.2 shows the fuel rate as a function of the total resistance acting on the ship and the ship's speed through water, respectively. Figure 5.3 shows the fuel rate with both dependencies at once.

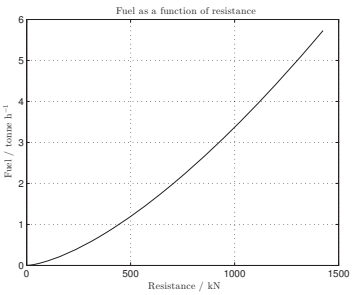


Figure 5.1: Fuel consumption as a function of total resistance.

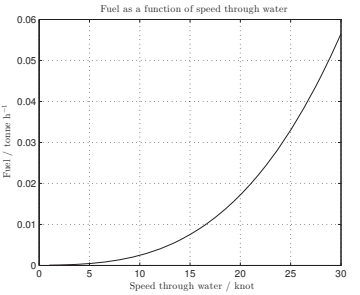


Figure 5.2: Fuel consumption as a function of speed through water.

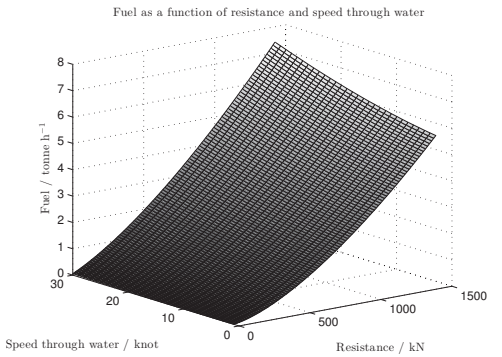


Figure 5.3: Fuel consumption as a function of total resistance and speed through water.

The total resistance is in turn modeled as the sum of the calm water resistance and the forces from the ocean waves acting on the ship in the longitudinal direction (also known as *surge*, the "front-back" motion). Figure 5.4 shows the calm water resistance as a function of speed through water, while Figure 5.5 shows the added wave resistance as a function of wave frequency (it is of course also dependent on other factors such as wave direction, wave height and more, but we will not go into any further details). We will use real wave data in the modeling.

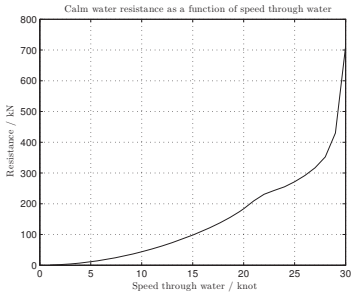


Figure 5.4: Calm water resistance as a function of speed through water.

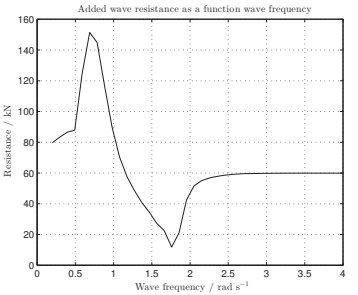


Figure 5.5: Added wave resistance in the longitudinal direction as a function of wave frequency.

The last objective function is regarding the "safety" of the journey. This will modeled as $f_3(\boldsymbol{x})$ = "maximum wave height", that is, the maximal wave height encountered in the whole route. For safety reason we therefore want to minimize this criteria.

5.2 The constraint functions

We will have four inequality constraints of the form

$$g_i(\boldsymbol{x}) \leq 0, \quad i = 1, \ldots, 4.$$

They are the following:

Distance $g_1(\boldsymbol{x})$
When moving from one point to another, we have a maximum distance constraint.

Land $g_2(\boldsymbol{x})$
We need to check for land when moving between two points. g_2 is the number of land crossings. It is calculated as numbers of "blocks" of land encountered in the route. Obviously $g_2 = 0$ for a feasible route.

Fuel rate $g_3(\boldsymbol{x})$
The ship has a maximum rate at which the fuel can be consumed.

Arrival time $g_4(\boldsymbol{x})$
The arrival time, t_f, should be reasonable. The constraint is

$$t_0 - \Delta t \leq t_f \leq t_0 + \Delta t$$

where t_0 is some nominal time and Δt is the time span.

5.3 Test voyages

There will be three different voyage planning problems that will be considered:

Gothenburg – Dunkirk (GD)
The voyage begins in Gothenburg, Sweden and ends in Dunkirk, France, see Figure 5.6. As one can see from the figure, this problem is characterized by tight land passages (land is marked as gray). Further, there are no ocean waves modeled in this problem, i.e. f_3 is always zero.

St. John's – Bodo (SB)
The voyage begins in St. John's, Canada and ends in Bodo, Norway, see Figure 5.7. There is a great amount of open spaces in this problem.

Gothenburg – New York (GN)
The voyage begins in Gothenburg, Sweden and ends in New York, USA, see Figure 5.8. It can be seen as a combination of GD and SB, with both tight passages and wide open spaces.

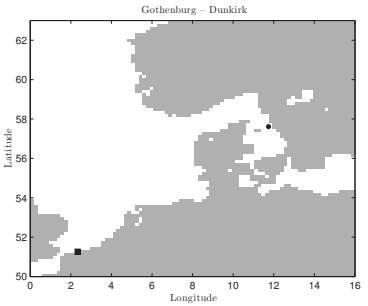


Figure 5.6: Problem GD. From Gothenburg (circle) to Dunkirk (square).

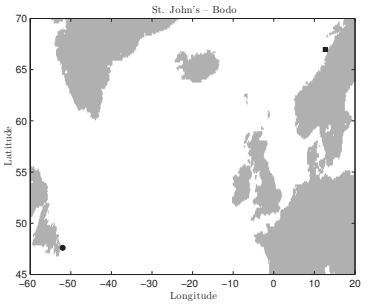


Figure 5.7: Problem SB. From St. John's (circle) to Bodo (square).

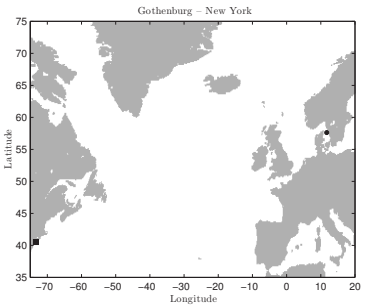


Figure 5.8: Problem GN. From Gothenburg (circle) to New York (square).

6 | Methods

There are two different main methods that will be used to solve the test problems: *Grid search* and *continuous search with DIRECT*.

6.1 Grid search

We will use the multi-objective optimization grid search method of [1] as described in subsection 4.2.2. The constraint g_4 (feasible arrival times) can been seen as a "translation" into the time points in the grid. A summary is given in algorithm 6.1 below.

Algorithm 6.1 Grid search

create a grid around a nominal route
$O \leftarrow$ departure node ▷ initialize set of optimal node combinations
for $k \leftarrow 1$ to number of stages **do**
    $Z \leftarrow \emptyset$ ▷ initialize empty set of objective vectors
    for all $d \leftarrow$ nodes in stage $k+1$ **do** ▷ the destination nodes
        for all $s \leftarrow$ nodes in stage k also in O **do** ▷ the source nodes
            if no constraints are violated when moving from s to d **then**
                $\boldsymbol{z}_{sd} \leftarrow$ calculate the objective functions when moving from s to d
                $Z \leftarrow Z + \boldsymbol{z}_{sd}$ ▷ add the objective vector to the set
            end if
        end for
    end for
    obtain the Pareto front in Z ▷ with either of the methods described above
    $O \leftarrow O +$ nodes d corresponding to the Pareto optimal points
end for

6.2 Continuous search with DIRECT

The other method is a continuous search using the DIRECT algorithm. The idea is to solve a scalarized problem of (2.1) using DIRECT. We begin by describing the decision variable $\boldsymbol{x}$ in $\boldsymbol{f}(\boldsymbol{x})$ as previously promised in chapter 5.

The method starts by gridding around a nominal route. As in Figure 4.2 we have a number of stages, however, there will not be any points extending orthogonally from the route. Instead, we have a continuous variable measuring this distance from the nominal route. That is, for stage k we have a variable x_k such that

$$-\delta \leq x_k \leq \delta, \quad \text{for } k = 2, \ldots, n-1,$$

where $\delta > 0$ is half the thickness of the grid around the nominal route and n is the number of stages in the grid (starting from the destination node $k = 1$ to the arrival node n), see Figure 6.1. We will thus have $n-2$ numbers of continuous position variables in the grid.

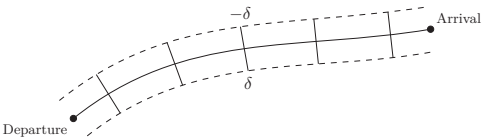


Figure 6.1: A nominal route gridded with a continuous search space along the orthogonal extensions. The position x_k in stage k can vary between $-\delta$ and δ, with $x_k = 0$ meaning a point on the nominal route.

For a stage k we will also have a variable v_k describing the speed used to reach a point in the next stage $k+1$, with the bounds

$$v_{\min} \leq v_k \leq v_{\max}, \quad \text{for } k = 1, \ldots, n-1,$$

where $v_{\min}$ and $v_{\max}$ is the lower respective upper speed bound of the ship.

The decision variable $\boldsymbol{x}$ in the objective and constraint functions is now

$$\boldsymbol{x} = (x_2, \ldots, x_{n-1}, v_1, \ldots, v_{n-1}),$$

with the dimension $2n-3$. With the objective and constraint functions from the previous chapter and with the variable bounds above, we have described the problem fully as a multi-objective optimization problem (2.1).

It will now be scalarized with the direction method. Let $t = h(\boldsymbol{l}, \boldsymbol{z})$ denote the scalarized function. That is, for a specific search line $\boldsymbol{l}$ and an objective vector $\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{x})$, $h(l, \boldsymbol{z})$ gives the t-value in (2.4). We then use DIRECT to minimize $h(\boldsymbol{l}, \boldsymbol{z})$, i.e. the t-value along the line.

But, what about the the constraint functions? DIRECT can not handle those. For this we use the penalty method, i.e. we will add a penalty when a constraint is violated

$$\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{x}) + \left(\sum_j \mu_j \max(g_j(\boldsymbol{x}), 0) \right) (1, \ldots, 1),$$

where $\mu_j > 0$ are penalty weights. This ensures that the objective function $h(\boldsymbol{l}, \boldsymbol{z})$ gets worse if some $g_i > 0$, because we then "move" the point $\boldsymbol{z}$ in the direction $(1, \ldots, 1)$ and thus obtain a greater t-value.

Algorithm 6.2 below concludes this method.

Algorithm 6.2 Continuous search using DIRECT

1: create a grid around a nominal route
2: $O \leftarrow \emptyset$ ▷ initialize optimal set
3: **for all** $\boldsymbol{l} \leftarrow$ search lines in the direction method **do**
4: setup the scalarized function $h(\boldsymbol{l}, \boldsymbol{z})$ as described above ▷ $\boldsymbol{z} = \boldsymbol{f}(\boldsymbol{x})$
5: $\boldsymbol{x}^* \leftarrow$ minimize $h(\boldsymbol{l}, \boldsymbol{z})$ with DIRECT ▷ algorithm 3.1
6: $O \leftarrow O + \boldsymbol{x}^*$
7: **end for**

7 | Results

In both methods the nominal route has been obtained with the isochrone method. In all problems, the grid has 8 nodes on each side of the nominal route with the distance 8 nautical miles between nodes in the same stage. Furthermore, the constraint on the arrival time has a window span of 12 hours of the nominal time.

7.1 Grid search

The following results was obtained by using 4 lines in the direction method. The last stage of the grid used 10 lines instead.

7.1.1 Gothenburg – Dunkirk

Figure 7.1 shows the nominal route for GD with the grid bounds.

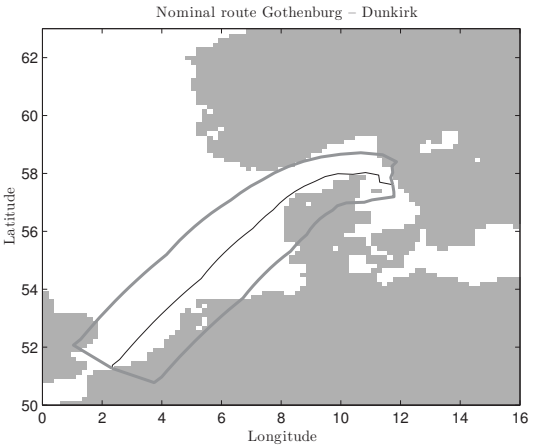


Figure 7.1: Nominal route (black) and grid bounds (gray). The grid has 39 stages.

Figure 7.2 shows the 91 Pareto optimal solutions found (spatially, they are all same).

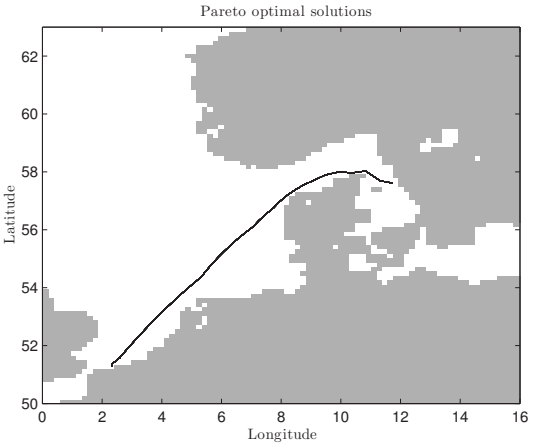


Figure 7.2: The Pareto optimal solutions.

Figure 7.3 shows the obtained Pareto front.

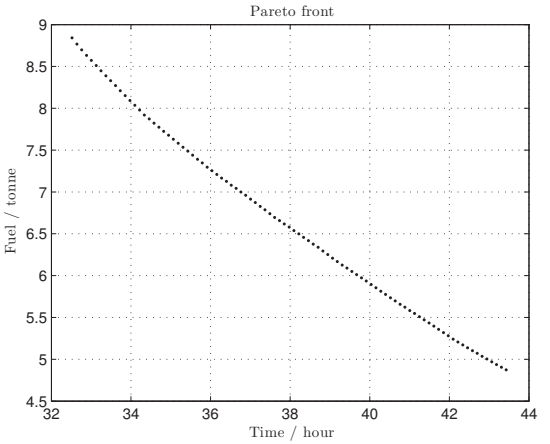


Figure 7.3: The Pareto front.

Figure 7.4 shows the speed over ground for the single-objective solutions.

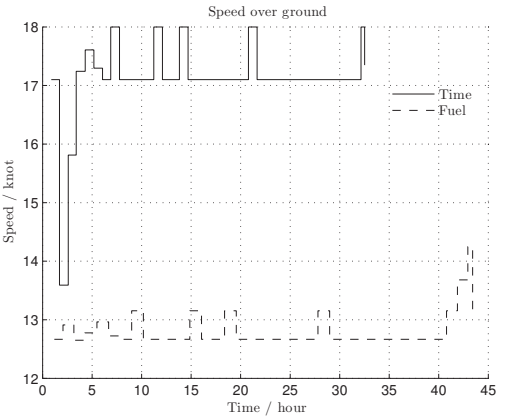


Figure 7.4: Speed over ground for the singe-objective solutions.

7.1.2 St. John's – Bodo

Figure 7.1 shows the nominal route for SB with the grid bounds.

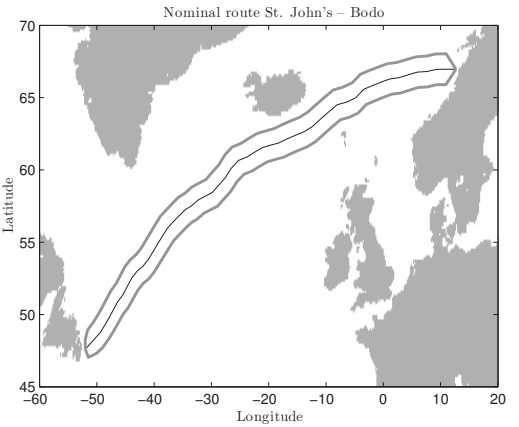


Figure 7.5: Nominal route and grid bounds. The grid has 54 stages.

Figure 7.6 shows the 490 Pareto optimal solutions found.

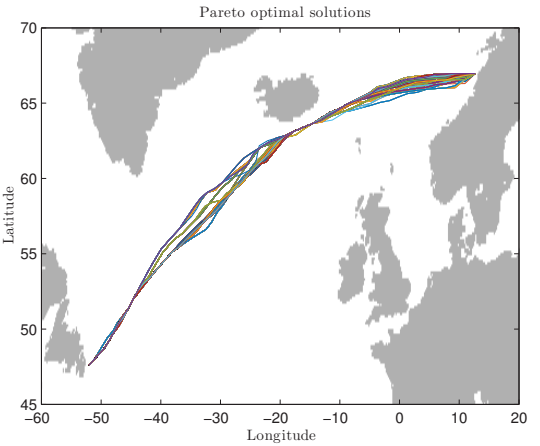


Figure 7.6: The Pareto optimal solutions.

Figure 7.7 shows the obtained three-dimensional Pareto front. Figure 7.8, Figure 7.9 and Figure 7.10 shows different projections of this Pareto front.

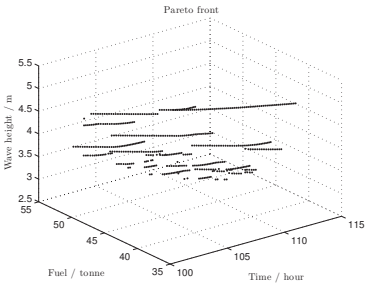


Figure 7.7: The Pareto front.

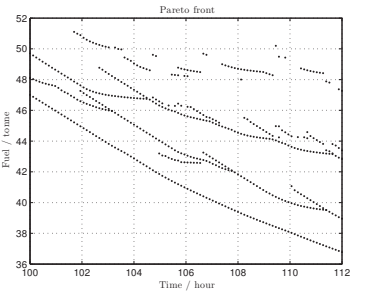


Figure 7.8: A different view of the three-dimensional Pareto front.

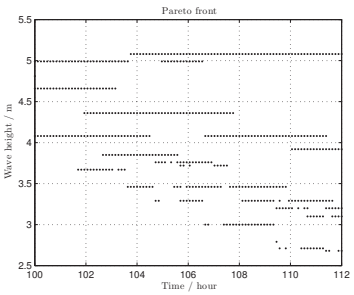


Figure 7.9: A different view of the three-dimensional Pareto front.

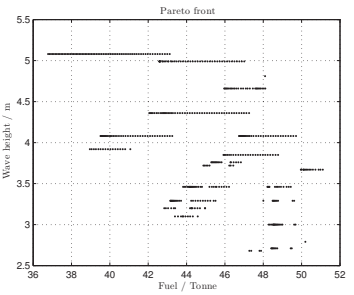


Figure 7.10: A different view of the three-dimensional Pareto front.

Figure 7.11 shows the single-objective solutions.

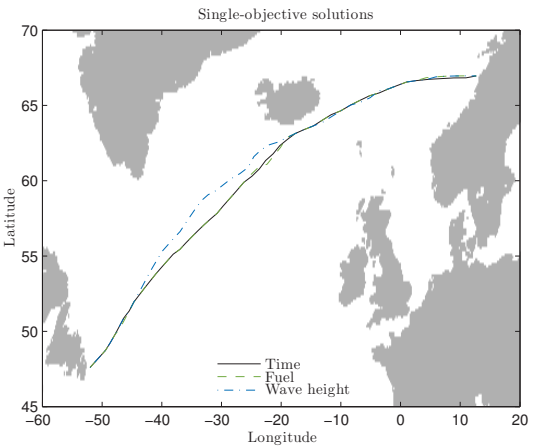


Figure 7.11: The single-objective solutions.

Figure 7.12 and Figure 7.13 shows the speed over ground and through water for the single-objective solutions, respectively.

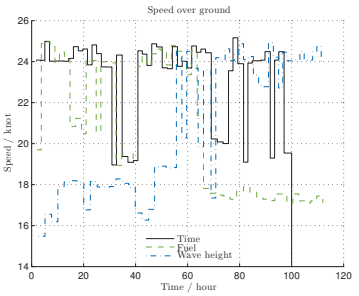


Figure 7.12: Speed over ground for the singe-objective solutions.

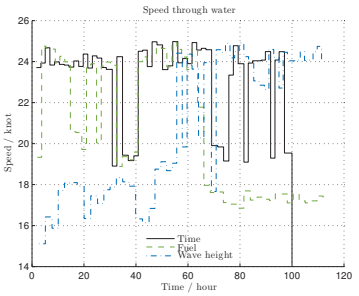


Figure 7.13: Speed through water for the singe-objective solutions.

7.1.3 Gothenburg – New York

Figure 7.1 shows the nominal route for GN with the grid bounds.

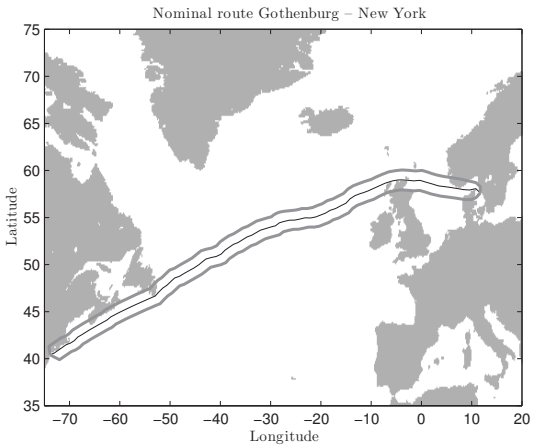


Figure 7.14: Nominal route and grid bounds. The grid has 90 stages.

Figure 7.15 shows the 383 Pareto optimal solutions found.

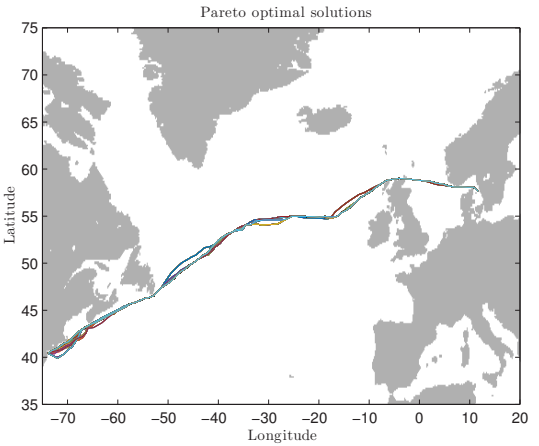


Figure 7.15: The Pareto optimal solutions.

Figure 7.16 shows the obtained three-dimensional Pareto front. Figure 7.17, Figure 7.18 and Figure 7.19 shows different projections of this Pareto front.

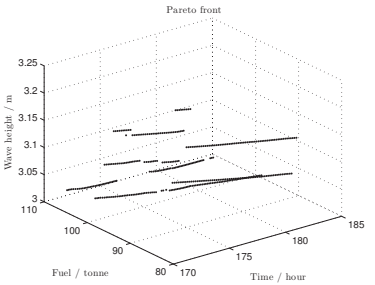


Figure 7.16: The Pareto front.

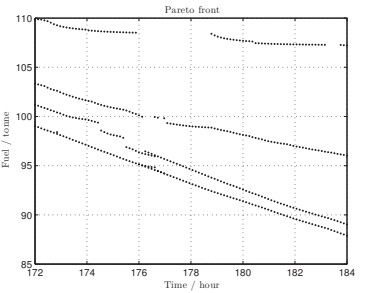


Figure 7.17: A different view of the three-dimensional Pareto front.

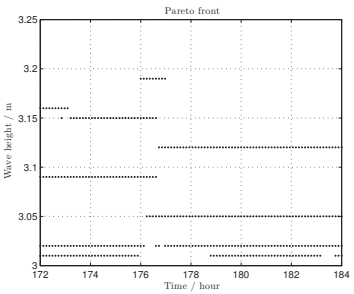


Figure 7.18: A different view of the three-dimensional Pareto front.

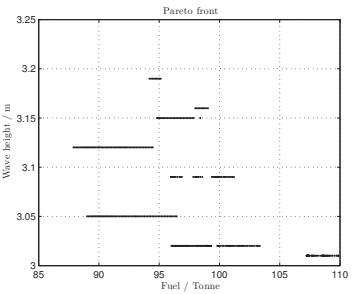


Figure 7.19: A different view of the three-dimensional Pareto front.

Figure 7.20 shows the single-objective solutions.

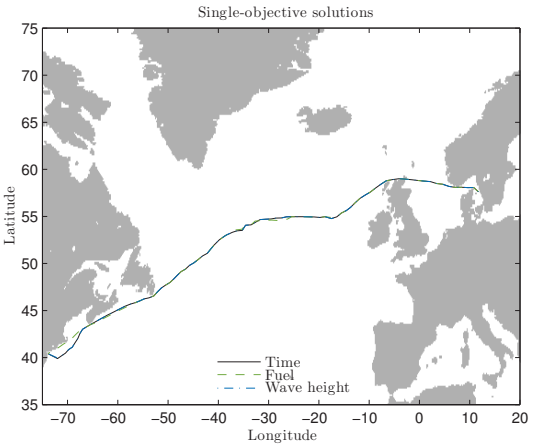


Figure 7.20: The single-objective solutions.

Figure 7.21 and Figure 7.22 shows the speed over ground and through water for the single-objective solutions, respectively.

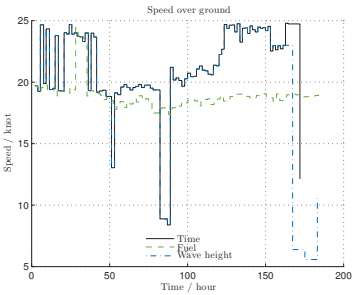


Figure 7.21: Speed over ground for the singe-objective solutions.

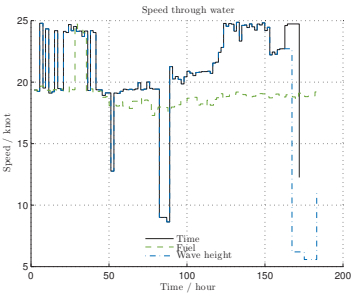


Figure 7.22: Speed through water for the singe-objective solutions.

7.1.4 Implementation aspects

Using the filtering and sample algorithm, without marking the dominated points (cf. section 2.5.2) and implemented in MATLAB, results in a computation time of 134.8 minutes for SB on 16 cores[1]. In the algorithm 3 points were sampled in each stage. In the last stage 10 points were sampled instead.

By marking the dominated points and implementing the Pareto filter in a compiled programming language such as C results in a computation time of 2.4 minutes with the same number of cores and sampled points.

Using the direction method instead, implemented in C, results in a computation time of 1.9 minutes for SB on 10 cores. The exact same implementation in MATLAB results a computation time of 3.0 minutes for SB on 10 cores. In the algorithm, 4 search lines was used in each stage. The last stage of the grid used 10 search lines instead. In comparison, filtering and sampling, implemented in C, takes 2.2 minutes for SB on the same number of cores.

Using 10 search lines in each stage instead, with the direction method implemented in C, gives a computation time of 3.6 minutes for SB on 10 cores.

Parallelization

Figure 7.23, Figure 7.24 and Figure 7.25 shows the speedup (time for one core / time for many numbers of cores) for the different problems. The parallelization is done for the loop on line 5 in algorithm 6.1. For all plots the direction method with 4 search lines was used in each stage. Again, the last stage of the grid used 10 search lines instead.

[1] Distributed on two Intel Xeon CPU E5-2690 (2.90 GHz).

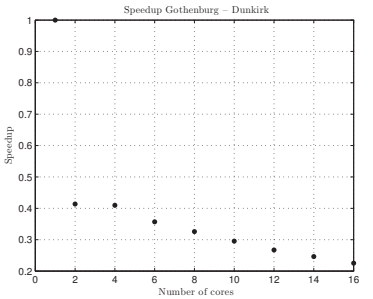


Figure 7.23: Speedup for GD.

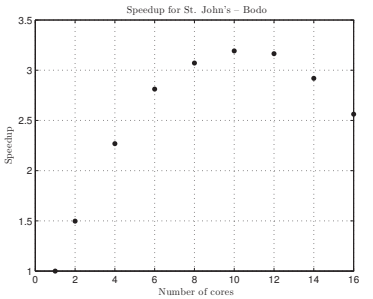


Figure 7.24: Speedup for SB.

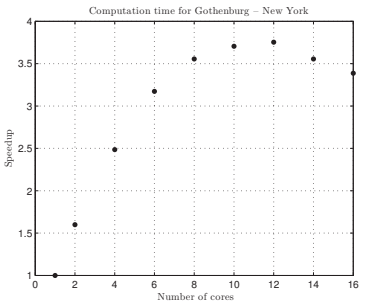


Figure 7.25: Speedup for GN.

7.2 Continuous search with DIRECT

The position variables have the bounds $[-64, 64]$ (same size as the grid search which had 8 points with 8 nautical miles between them). The speed variables have the bounds $[5, 18]$ (in knots).

The stopping criteria was set to 50000 function evaluations for DIRECT. 4 search lines was used in the direction method.

Figure 7.26 shows the 4 Pareto solutions found for GD.

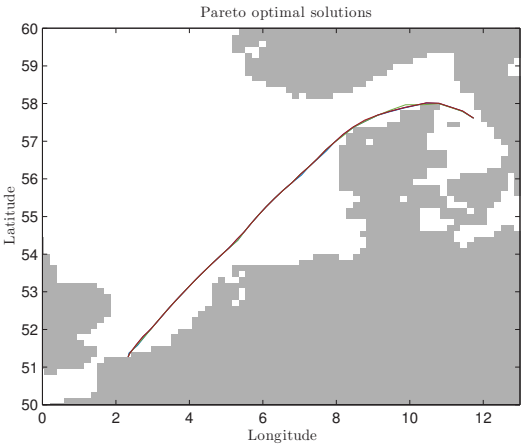


Figure 7.26: The Pareto optimal solutions.

The single-objective solutions obtained was $(f_1, f_2) = (32.2 \text{ hours}, 9.1 \text{ tonnes})$ for f_1 and $(f_1, f_2) = (43.4 \text{ hours}, 5.0 \text{ tonnes})$ for f_2 (from Figure 7.3, the grid search solutions for GD was 32.5 hours for f_1 and 4.9 tonnes for f_2).

Figure 7.27 shows the obtained Pareto front for GD. Figure 7.28 shows a comparison between the Pareto fronts obtained with DIRECT and grid search.

Figure 7.29 shows the speed over ground for the Pareto solutions for GD.

A stopping criteria of 50000 function evaluations results in a running time of approximately 30 minutes for the DIRECT algorithm. Finding the two single-objective solutions for GD (recall that $f_3 = 0$ for this problem) takes therefore $2 \cdot 30 = 60$ minutes (used for scaling, explained in the next chapter). With additionally 4 search lines this means a total running time of $60 + 4 \cdot 30 = 180$ minutes.

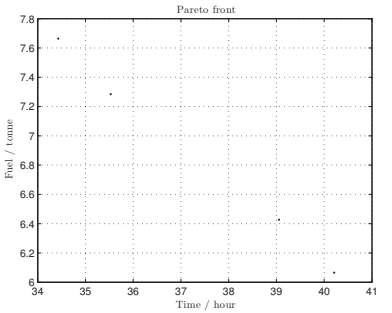


Figure 7.27: The Pareto front.

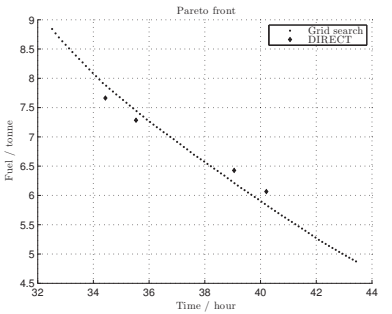


Figure 7.28: A comparison of the Pareto fronts obtained for GD with DIRECT and grid search.

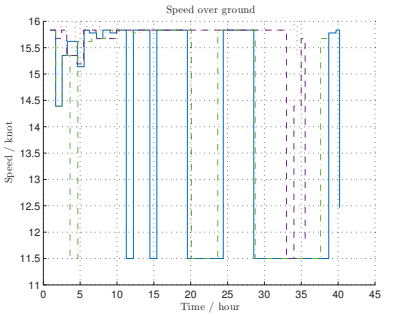


Figure 7.29: Speed over ground for the Pareto solutions.

8 | Discussion & conclusions

8.1 Grid search

As one can see from Figure 7.3, Figure 7.7 and Figure 7.16 the obtained Pareto front is rather well spread, even though only 4 search lines was used. The conflict between the arrival time and fuel consumption is also evident.

8.1.1 Implementation aspects

From Figure 7.24 and Figure 7.25 we see that parallelization may reduce the computation time substantially. The minimum time occurs around 10 cores for SB and 12 cores for GN. The reason why the computation time increases after these numbers is simply because of the overhead. The more cores, the more data needs to be transferred. Thus after some number of cores the overhead will be greater than the actual time running the loop. For GD, this already occurs for two cores, see Figure 7.23. It is thus not even advantageous to parallelize this short route.

However, by saving constant data on a static shared memory instead of sending it repeatedly to the cores during the parallelization, a brief testing resulted in roughly twice as fast computation time for GD. For the other problems, SB and GN, no significant improvement was observed.

One clearly also gains much by implementing the Pareto filter algorithm more efficiently, and in a compiled language. This results in a 98 % reduction of the running time! Moreover, the direction method is slightly faster than filtering and sampling when both are implemented in C. However, when implemented in MATLAB the difference in running time is huge, again a reduction of 98 % is obtained. This might be expected, since unlike first filtering and then sampling the Pareto front, the direction method finds and samples in one go. The reason why there is a greater improvement seen in the implementation in MATLAB is the following: Both algorithms in C are very efficiently implemented. Thus, for relatively "small" problems the bottleneck is somewhere else in the whole program. The same huge difference would probably have been experienced for the algorithms implemented in C if one used a sufficiently large problem, so that the bottleneck happens to be exactly the sampling algorithms.

A consequence of not knowing the Pareto front beforehand is not being able to scale exactly as in (2.2). Equation (2.2) requires the nadir point, which we obviously have no information of without knowing the Pareto front (see definition 2.3). Therefore, in the actual implementation we scale with the single-objective solutions instead. In higher dimensions than two, this is obviously not totally safe, as some points on the Pareto front may be scaled incorrectly and thus fall outside the range $[0, 1]$. The alternative would be to scale with the worst obtainable value instead of the nadir point, but then several search lines might actually miss the Pareto front.

A special case in the direction method, that we did not mention, is if two or more points happens to have the same t-value. That is, if two or more points happens to belong on the boundary shown in Figure 2.9. One might naively think that this is numerically very improbable. Because, for this to actually happen, the objective vectors needs to have at least one coordinate with the exact same value. However, recall that one of the coordinates is the value of the objective function f_3, which is the maximum wave height. This value can stay the same for a great region of the ocean. The case is thus not very improbable, and has actually been encountered through the calculations.

How did we solve it? We did not. If two or more points had the same t-value, we just returned one of them, even if it was not Pareto optimal. The expectation was that another search line might sort the situation out by picking the correct optimal point. However, we can not leave everything to chance. Therefore, a thorough Pareto filtering was done once on the final solutions, filtering out the false picked non-optimal points (however some optimal solution might already have been lost).

In fact, because of this, the third objective function f_3 might not be such a good choice for safety and comfort. It might have been better as a constraint instead. For instance, some kind of mean value of the wave heights, or some other measurement, might have been better.

8.2 Continuous search with DIRECT

In theory, DIRECT should be able to find a better optimum than grid search. This is because, unlike grid search, we do not search over discrete sample points but in a continuous search space. That is, if the optimal solution happens to lie between any sample points (which likely is the case), grid search will not be able to find it. Figure 7.28 shows an example of this. Two of the four solutions obtained with DIRECT are better. However, the points are not equidistantly spread out even though the search lines are put equidistantly on the reference plane.

Clearly DIRECT is not very computationally effective as compared to grid search. The reason is that the algorithm is global; Even if DIRECT confines into a small area in the search space, it still continues to evaluate the whole search space (cf. Figure 3.4). Just to guarantee a global optimum.

Moreover, a stopping criteria is hard to specify for DIRECT. The first time we run DIRECT is to find the single-objective solutions for scaling, as described in the previous section. Thus, it is desired that these points are relatively close to the correct single-objective solutions. But we do not know these values beforehand, and a stopping criteria based on that is therefore useless. We can thus only hope that DIRECT finds the correct ones after a number of function evaluations. Also, we can not stop if the minimum value stagnates sometime during the search. This might just be a temporary "false" plateau.

When choosing an algorithm to minimize along the lines, we obviously require a gradient-free method, which DIRECT is. The reason the DIRECT algorithm was chosen, was based on a previous promising voyage planning result [11]. However, that result was based on a 13-dimensional problem. Our "smallest" problem GD consists of $2 \cdot 39 - 3 = 75$ dimensions! Clearly DIRECT is not suitable for very high-dimensional problems.

As previously said, DIRECT assumes a continues objective function. Since we assign a penalty when a constraint is violated, this gives rise to discontinuities. However, this will not affect DIRECT in the minimization procedure, because this type of discontinuity only yields positive "spikes" in the objective function. It will therefore not interfere with a minimum in a "deep valley".

8.3 Future work

Although highly acceptable computation times and results have been obtained with the grid search, the computation time might be further reducible. For instance, putting the parallelization elsewhere in the grid search algorithm could reduce the time additionally. A thorough investigation concerning this should therefore be carried out. Another idea that might reduce the time even further is saving some constant data on a static shared memory instead of sending it repeatedly to the cores during the parallelization. As previously said, this resulted in faster computation time for GD.

The continuous search methods should not be totally ruled out. Another algorithm might give better performance. For instance, a few short tests have been made with the algorithm COBYLA instead of DIRECT, with quite promising results.

8.4 Conclusions

Although the grid search method is approximative, it can be used in practical multi-objective voyage planning optimization problems, with satisfying results. Using parallelization one can improve the computational time significantly, making a multi-objective optimization practically feasible.

It is not suitable to use continuous search method with DIRECT for solving multi-objective realistic voyage planning optimization problem. However, substituting DIRECT with another algorithm might yield more promising results.

When obtaining the Pareto front, the direction method is much faster than filtering and sampling. The reason being, in the direction method one both filters and samples in one go. In the evaluated test cases the difference in time is quite small when the routine is implemented in C, but when implemented in MATLAB the difference is huge.

Bibliography

[1] Andersson, A. (2015). *Multi-objective optimisation of ship routes*. Gothenburg: Chalmers University of Technology.

[2] Bijlsma, S. J. (1975). *On Minimal-Time Ship Routing*. Delft University of Technology.

[3] Bleick, W. E & Faulkner, F. D. (1965). *Minimal-Time Ship Routing*. Journal of Applied Meteorology. Volume 4, 217–221.

[4] Chen, H. (1978). *A Dynamic Program for Minimum Cost Ship Routing under Uncertainty*. Massachusetts Institute of Technology.

[5] Deb, K, Agrawal, S., Pratap, A. Meyarivan, T. (2002). *A fast and elitist multiobjective genetic algorithm: NSGA-II*. IEEE Transactions on Evolutionary Computation. Volume 6, 182–197.

[6] Hagiwara, H. (1989). *Weather routing of (sail-assisted) motor vessels*. Delft University of Technology.

[7] Haltiner, G. J., Hamilton H. D., & Arnason G. (1962). *Minimal-Time Ship Routing*. Journal of Applied Meteorology. Volume 1, 1–7.

[8] Hinnenthal, J. (2007). *Robust Pareto-Optimum Routing of Ships utilizing Deterministic and Ensemble Weather Forecasts*. Technical University Berlin.

[9] James, R. W. (1957). *Application of wave forecast to marine navigation*. Washington: US Navy Hydrographic Office.

[10] Jones, D. R., Perttunen C. D., & Stuckman B. E. (1993). *Lipschitzian Optimization Without the Lipschitz Constant*. Journal of Optimization Theory and Application. Volume 79, 157–181.

[11] Larsson, E., & Simonsen, H. M. (2014). *DIRECT Weather Routing*. Gothenburg: Chalmers University of Technology.

[12] Miettinen, K. (1999). *Nonlinear Multiobjective Optimization*. Kluwer Academic Publishers.

[13] Nordström, P. (2014). *Multi-objective optimization and Pareto navigation for voyage planning*. Uppsala University.

[14] Pascoletti, A. & Serafini, P. (1984). *Scalarizing vector optimization problems*. Journal of Optimization Theory and Applications. Volume 42, 499–524.

[15] Shubert, B. (1972). *A Sequential Method Seeking the Global Maximum of a Function*. SIAM Journal on Numerical Analysis. Volume 9, 379–388.

[16] Skoglund, L., Kuttenkeuler, J., & Rosén, A. (2012). *A new method for robust route optimization in ensemble weather forecasts*. Stockholm: Royal Institute of Technology.

[17] Szlapczynska, J., & Smierzchalski R. (2007). *Adopted isochrone method improving ship safety in weather routing with evolutionary approach*. International Journal of Reliability Quality and Safety Engineering. Volume 12.

[18] Zitzler, E.& Thiele, L. (1999). *Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach*. IEEE Transactions on Evolutionary Computation. Volume 4, 257–271.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER´S THESIS

Date of issue
December 2015

Document Number
ISRN LUTFD2/TFRT--5994--SE

Author(s)
Waqar Hameed

Supervisor
Mats Molander, ABB
Pontus Giselsson, Dept. of Automatic Control, Lund University, Sweden
Giacomo Como, Dept. of Automatic Control, Lund University, Sweden (examiner)

Sponsoring organization

Title and subtitle

Multi-Objective Optimization of Voyage Plans for Ships

Abstract

In this thesis two methods are investigated to solve a multi-objective optimization problem for voyage planning. The first method, grid search, is a brute force search in a three-dimensional graph while the other uses the Lipschitzian algorithm DIRECT to do a continuous search along a nominal route.

The grid search method gives a computation time of 7.6 minutes for a route from Gothenburg to New York. This is obtained partly by parallelizing on 10 cores but also implementing core routines efficiently in compiled programming languages.

However, the continuous search method with DIRECT is not suitable for a realistic voyage planning problem. It is more due to the nature of the DIRECT algorithm than the implementation details.

Keywords
multi-objective optimization, voyage plan, Pareto sample, direction method, grid search, Lipschitzian optimization, DIRECT

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-52

Recipient's notes

Security classification

http://www.control.lth.se/publications/