DELFT UNIVERSITY OF TECHNOLOGY

REPORT 19-04

Determining water speed of ships: Establishing the delivered power needed as a function of the ship's speed in relation to the water

M. Atsma and C. Vuik

Contents

Preface

This report describes the research that started with the research of Floris Buwalda and continued with the work of Michaël Mersie. We will mostly work with Numerical Mathematics to find a function that expresses the power that needs to be delivered in terms of the ship's desired speed in relation to the water. In order to keep up with the digital age where contracts and transactions are calculated to fractions of cents, it is essential that this function is as accurate as possible.

We would like to take this opportunity to thank Hans Huisman and MARIN for their time and for contributing necessary data to this research.

1

Introduction

With the rise of computers, the calculations in the economy are becoming more and more accurate. In this multiple decimal economy, ships have been left behind, even though they play a major role in it. In many cases, a classical method is still used to calculate the power consumption needed for a certain speed. In this dissertation, we will discuss this method and newer ones.

Previously, Floris Buwalda[1] and Michaël Mersie[2] have studied this subject and made some improvements. We will aim to explain, and work on, the accuracy of these newer methods. In addition, we will bring up problems that the classical method has, which is eliminated by Mersie's new method.

The main questions we will answer in this report are the following:

- Are there certain situations in which the Iterative Method is better than the Direct Method?

- Does an iteration of the Mean of Means Method reliably stop the Iterative Method from diverging?

This report has the following structure. After an introduction of the physics of the problem in Section 2 we give an overview of solution methods in Section 3. We combine the methods with measurements to see which method has the best properties. This research is reported in Section 4. Conclusions are given in Section 5, whereas the appendix contains the used Python Code.

2

Physics behind the problem

A ship traveling through water at a *constant* speed experiences resistance forces that should add up to equal the force that the engine produces. These forces come from both water and air. The total 'still water' resistance can be divided into three fundamental components as shown by Van Manen and Oossanen [3]:

- The frictional resistance, $R_{drag} = \frac{1}{2}S\rho V^2$, where

$$S : \text{wet frontal area of the ship in } \left[m^2\right]$$

$$\rho : \text{water density in } \left[\frac{kg}{m^3}\right]$$

$$V : \text{ship's velocity relative to the flow in } \left[\frac{m}{s}\right]$$

- The wave-making resistance, represented by the dimensionless Froude number, $Fr = \frac{V}{\sqrt{gL}}$, where

$$V : \text{relative velocity of the ship in } \left[\frac{m}{s}\right]$$

$$L : \text{waterline length of the ship in } [m]^1$$

$$g : \text{gravitational acceleration of } 9.81\frac{m}{s^2}$$

- The eddy making resistance, which, according to Tupper [4], is 10-15% of the hull resistance and thus not critical.

2.1. Dimensional analysis

A dimensional analysis can be done to determine an intuitive insight of what the function of the total resistance R will look like. This means that instead of a value,

we express variables in their characteristic terms (time T, mass M, and length L), then use these expressions to determine how many times we should multiply certain terms with each other, as in the end we want to have the same power of characteristic terms on both sides of the equation.

Tupper [4] states that the contributing factors are the density, ρ, viscosity, μ, and the static pressure in the fluid, p, as well as the velocity, V, the length of the ship, L, and the gravitational accelleration, g. Thus, we get that the resistance is some formula of those constants, i.e.

$$R = f(L, V, \rho, \mu, g, p) \tag{2.1}$$

The characteristic terms for these variables are

$$R : \frac{ML}{T^2} \qquad \rho : \frac{M}{L^3} \qquad \mu : \frac{M}{LT} \qquad g : \frac{L}{T^2}$$

According to Tupper, this can then be written as

$$R = \rho V^2 L^2 \left[f_1 \left(\frac{\mu}{\rho V L} \right), f_2 \left(\frac{gL}{V^2} \right), f_3 \left(\frac{p}{\rho V^2} \right) \right] \tag{2.2}$$

Thus, the following non-dimensional ratios are important:

- Resistance coefficient: $Rc = \frac{R}{\rho V^2 L^2}$

- Reynold's number: $Re = \frac{VL\rho}{\mu}$
  A low Reynold's number indicates laminar flow, while a high Reynold's number indicates turbulence.

- Froude number: $Fr = \frac{V}{\sqrt{gL}}$
  This represents the ratio of the flow inertia to the external field. Vessels with the same Froude number produce the same wake, even if their size and shape are different.

- Euler number: $Eu = \frac{p}{\rho V^2}$
  This characterises the energy losses in the flow. A frictionless flow corresponds to a flow with an Euler number equal to 1.

So from the dimensional analysis we gather that the resistance comes from certain dimensionless quantities. Therefore, other than knowing the frictional resistance is a function of these Rc, Re, Fr, and Eu, we do not gain more knowledge.

3

Solving methods

A *speed trial* is when a ship sails either in the exact direction of the wind, or in its exact opposite direction. It must also sail on a predetermined power setting, for a certain amount of time. Lastly, there should be limited wind, waves, and currents, and it should ideally be free from hindrance of small boats and commercial traffic. The speed trial is split up into smaller time fragments (all equal, with a minimum of 10 minutes) that we will call *"Runs"*. At the end of a run, wherever the ship is at that time will be the starting point for the next Run, after the ship has turned around — to sail in the reciprocal direction — and reset the power setting. Any deviances in the power setting that occur during a Run are not adjusted, as fluctuations are worse for the accuracy of the measurement than slight deviations in the power settings are. The term *"Double Run"* is used to group two successive Runs in opposite directions. Each Run will result in averaged values that can be used in calculations:

- The *timestamp*, t, when a Run has taken place is the time halfway through the Run.

- The *delivered power*, P, of a Run is the average power delivered during the Run.

- The *ground speed* of a Run — that is to say the speed of the ship in relation to its geographical position, and *not* in relation to the water — is the average ground speed of the Run

There are different ways to use this data to determine the speed of the ship. Discussed are the *Mean of Means*, *Iterative*, and *Direct* methods.

3.1. Mean of Means Method

With the Mean of Means method, we take the data of all the Runs and take the average of successive Runs. These new averages have 1 less data point than the
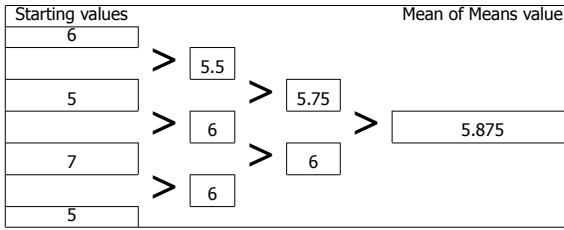
Table 3.1: Small example of a Mean of Means calculation

3

previous values. We repeat this until there is only one value left. This is the Mean of Means. An illustrative example can be seen in Table 3.1.

This computation can be expressed in a single formula. If we have n values for the averaged measured speeds V_i, then the Mean of Means value V_{MoM} can be expressed as

$$V_{MoM} = \frac{1}{2^{n+1}} \sum_{i=1}^{n+2} \binom{n+1}{i-1} V_i.$$ (3.1)

If the speed of the current, V_C, is constant during the speed trial, one can see how every Double Run the contribution of the current will cancel out. Unfortunately, one could see how the deviations can become too big when it takes a ship over an hour to turn around. Therefore, we rely on an n-th degree polynomial approximation of $V_C(t)$. Thus we can say

$$V_C(t) = \sum_{j=0}^{n} a_j t^j$$ (3.2)

for some numbers (a_j).

Mersie shows that we then need (at least) $n+2$ Runs, and he notes that usually n is chosen to be 2, 3 or 4. This is because the longer the speed trial takes, the less accurate the polynomial approximation of V_C becomes.

If we assume V_S to be the actual speed of the ship, $V_{G,i}$ to be the measured ground speed of the ith Run, and Δt to be the time each Run takes, then

$$V_{G,i} = V_S \pm V_{C,i}$$ (3.3)

$$= V_S + (-1)^{i+1} \sum_{j=0}^{n} a_j (i\Delta t)^j$$ (3.4)

because the contribution of the current speed changes sign when the ship sails in opposite directions[1].

---

[1]Note that an n-th degree polynomial function can be shifted to the right or left without changing its degree. Thus we can make sure that the value of $V_C(i\Delta t)$ corresponds to the current speed contribution at the halfway mark of the ith Run (as that is defined to be the timestamp of the Run).

By substituting (3.4) in (3.1) and working it out we get

$$
\begin{aligned}
V_{MoM} &= \frac{1}{2^{n+1}} \sum_{i=1}^{n+2} \left[\binom{n+1}{i-1} \left[V_S + (-1)^{i+1} \sum_{j=0}^{n} a_j (i\Delta t)^j \right] \right] \\
&= V_S + \frac{1}{2^{n+1}} \sum_{j=0}^{n} \left[a_j \Delta t^j \sum_{i=1}^{n+2} \binom{n+1}{i-1} (-1)^{i+1} i^j \right] \\
&= V_S + \frac{1}{2^{n+1}} \sum_{j=0}^{n} \left[a_j \Delta t^j \cdot 0 \right] \\
&= V_S.
\end{aligned}
\tag{3.5}
$$

3.2. Iterative Method

To improve this approximation of the tides, the Iterative Method has been developed. In this method, we consider the tidal current in a more accurate form. Floris Buwalda [1] has previously improved the form that was set by ISO [5] into the following

$$
V_C(t) = V_{C,c} \cdot \cos(2\pi\tau) + V_{C,s} \cdot \sin(2\pi\tau) + V_{C,t} \cdot \tau + V_{C,0}
\tag{3.6}
$$

where

$$V_C(t) : \text{current speed on time } t \text{ in } \left[\frac{L}{T}\right];$$

$$\tau : \text{the dimensionless variable given for } \frac{t}{T_C};$$

$$t : \text{time in } [T];$$

$$T_C : \text{period of the dominant tidal consituent in units of time, } [T], \text{ namely 12 hours, 25 minutes, and 12 seconds;}$$

$$V_{C,c}, V_{C,s}, V_{C,t}, V_{C,0} : \text{unknown constants.}$$

The unknown constants here are named as such so that they represent the contributions to the speed of the current. The second letter in the subscript corresponds to, respectively, the **C**osine, **S**ine, linear **t**ime, and a constant (**0**). With the Iterative Process we will then try to approximate these unknown constants.

Eventually, we want to figure out the relation between the delivered power (P) and the ship's speed relative to the water (V_S). Firstly, it is assumed that the following relation holds:

$$
P(V_S) = a + b \cdot (V_S)^q
\tag{3.7}
$$

or equivalently:

$$V_S = \left(\frac{P(V_S) - a}{b} \right)^{\frac{1}{q}} \tag{3.8}$$

but we do not know what these a, b and q are.

By ISO's standards [5], a minimum of 4 Double Runs is needed. It will become clear why that is shortly. So that is 8 Runs. The power setting P for both Runs of a Double Run should be nearly equal. The power setting of the different Double Runs should be spread between 65% and 100% [6]. When we average the P and V_S of each Double Run, it will result in a (P, V_S) pair for each Double Run. Thus, we will have 4 (P, V_S) pairs, where P is a function of V_S as stated in (3.7).

Then, a 'least squares' fitting method is used on these 4 value-input-pairs to the function (3.7). This fitting will determine an approximation of a, b, and q, and thus a first approximation of the power/speed function is made. These are 3 variables that need to be fitted. As we will see later, in (3.6) we even have 4 unknown constants. One might now understand why a minimum of 4 Double Runs has been set.

Now, we have an initial approximation of the power/speed function. However, the V_S in these pairs that this fitting was based on, do not yet account for the current as given by (3.6). The problem there is that we also do not have the values for $V_{C,c}$, $V_{C,s}$, $V_{C,t}$, and $V_{C,0}$. So, we develop an iterative process to improve this approximation step by step. This will be done until our calculated Power is close enough to the measured Power.

We now need to consider for which constants the values are certain, and which ones depend on the formulas. Certain are the following:

- $P_{id,i}$, because the ship delivered a certain amount of power, which has been measured.

- $V_{G,i}$, the measured ground speed of Double Run i, because we have GPS.

- t_i, the timestamp of Double Run i

And the rest can only be determined by approximating the formulas.

To better follow the steps we will be taking, please refer to Figure 3.1.

The first step is to take the current $V_{S,i}$, a, b, and q, and set a value to P_i by formula (3.7). These are now the values of P_i according to our found variables. Next, we check if these P_i are close enough to the actual delivered power $P_{id,i}$, by the error function

$$E = \sum_i \left(P(V_S')_i - P_{id,i} \right)^2 \tag{3.9}$$

where i is the index of the Double Run, $P(V_S')$ the Delivered Power according to the current approximation of formula (3.7), and P_{id} is the actually delivered power with a number of corrections.
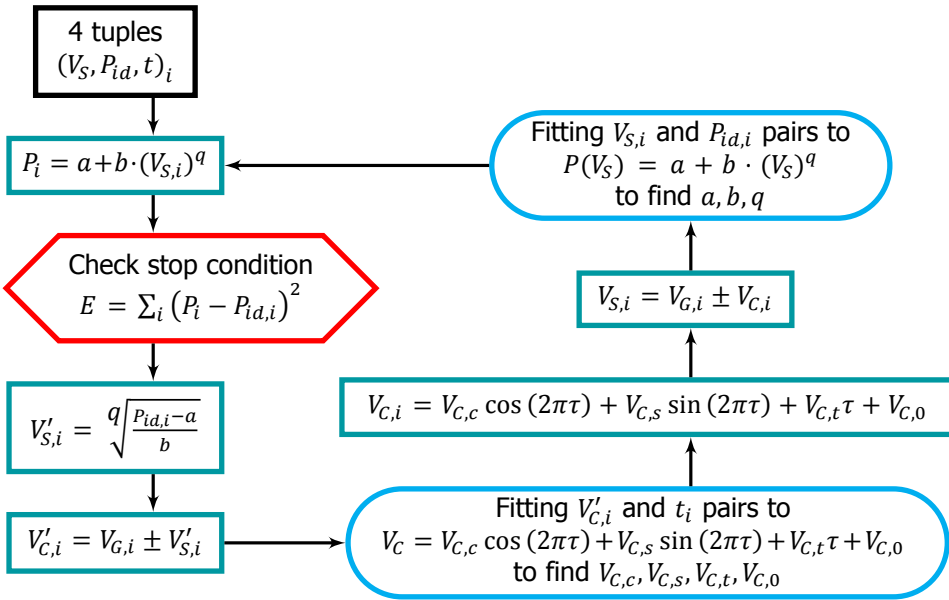
Figure 3.1: A visual representation of the Iterative Method. Note that in the green, rectangled nodes we set new values to variables; in the blue, rounded nodes we fit a function; and the stop condition is checked in the red, hexagonal node.

If we are not close enough yet, we set an intermediate value of $V'_{S,i}$ by formula (3.8) and our current values of P_i, a, b, and q. With this intermediate value we can calculate an intermediate value of $V'_{C,i}$ with $V'_{C,i} = V_{G,i} \pm V'_{S,i}$, because we know $V_{G,i}$ for certain and we have just created $V'_{S,i}$. We set this per Single Run, as the $\pm$ here depends on whether the ship is going up or downstream.

This $V'_{C,i}$ gives us a value for the current speed during the Run taking place at time t_i. With these values, we make a fitting of function (3.6) to find values for $V_{C,c}$, $V_{C,s}$, $V_{C,t}$, and $V_{C,0}$. This step will give us a complete (approximated) function of the current speed V_C according to time t.

With this function, we can set a value to $V_{C,i}$ by only having to insert the timestamp t_i of the Run. The function value of $V_{C,i}$ gives us a new value for $V_{S,i}$ with the function $V_{S,i} = V_{G,i} \pm V'_{C,i}$.

Now we are back to where we started, where we have values for $V_{S,i}$, and the measured values for $P_{id,i}$. So we can once again make a fitting of function (3.7) to find new values for a, b, and q. This, in turn, gives us new — hopefully improved — values for the function value of P_i. Once again, we reach the stop condition to check if our error is small enough now.

The Iterative Process means that we stay in this loop until the error is small enough. This, however, also raises a problem: will the method even converge? In fact, the existence of a limit of this Method has yet to be proven. Therefore, we assume that it exists. Additionally, even if the error converges to some value, it might not converge to 0. Luckily, any converging sequence is also a Cauchy

sequence. So we can look at two consecutive values of the error function, and check if that difference is small enough instead. So when $|E_n - E_{n-1}| < 10^{-4}$, we stop, which means we have come close to the limit of the error value[2].

Another "ambiguity" problem is the choice for the initial conditions. For any non-linear "least squares" fitting, we first need to estimate the correct values, regardless of the method. It has not yet been explored how the current estimate using the Mean of Means Method affects the fittings.

Lastly, there are various non-linear least squares fitting methods that can be used. Two methods, the Trust Region Reflective and the Levenberg-Marquardt Method, are explored later.

3.3. Direct Method

The Direct Method makes use of substitution, so we do not have to keep updating two different function approximations, and instead only need to do one fitting. We substitute (3.6) into (3.3), which we then substitute into (3.7) to get the following expression

$$P(V_G, \tau) = a + b\left[V_G \pm \left(V_{C,c}\cos\left(2\pi\tau\right) + V_{C,s}\sin\left(2\pi\tau\right) + V_{C,t}\tau + V_{C,0}\right)\right]^q \qquad (3.10)$$

where

$$V_G : \text{ground speed of the ship in } \left[\frac{m}{s}\right];$$

$$\tau : \text{dimensionless time, i.e. } \tau = \frac{t}{T_C}\ ;$$

$$a, b, q, V_{C,c}, V_{C,s}, V_{C,t}, V_{C,0} : \text{unknown constants to be fitted.}$$

For n Double Runs we have $2n$ Single Runs. So, we get the following system of equations

$$
\boldsymbol{P}(\boldsymbol{V_G}, \boldsymbol{\tau}) = \begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_{2n} \end{bmatrix}
$$

$$
= \begin{bmatrix} a + b\left[V_{G,1} \pm \left(V_{C,c}\cos\left(2\pi\tau_1\right) + V_{C,s}\sin\left(2\pi\tau_1\right) + V_{C,t}\tau_1 + V_{C,0}\right)\right]^q \\ a + b\left[V_{G,2} \pm \left(V_{C,c}\cos\left(2\pi\tau_2\right) + V_{C,s}\sin\left(2\pi\tau_2\right) + V_{C,t}\tau_2 + V_{C,0}\right)\right]^q \\ \vdots \\ a + b\left[V_{G,2n} \pm \left(V_{C,c}\cos\left(2\pi\tau_{2n}\right) + V_{C,s}\sin\left(2\pi\tau_{2n}\right) + V_{C,t}\tau_{2n} + V_{C,0}\right)\right]^q \end{bmatrix} \qquad (3.11)
$$

In this system, we have 7 constants that need to be fitted. That means that we need at least 4 Double Runs (which makes for 8 Runs), so that we have enough equations in our system. This also lines up with ISO's previously mentioned standard. It also means that the initial guesses need to be reasonably close to the true values. However, the advantage is that the power settings do not need to be the same anymore for the two Runs in a Double Run. And in addition, we need only to do one fitting.

---

[2]The value 10^{-4} is chosen to be reasonably small, but also large enough to be above computer precision.

3.4. Least squares fitting methods

There are several methods to do a least squares fitting. In this paper we will consider and compare two popular ones: Levenberg-Marquardt, and Trust Region Reflective. Before we elaborate more on these methods individually, we note what they try to achieve.

Let f be a non-linear function of $\boldsymbol{x}$ and $\boldsymbol{\beta}$, where $\boldsymbol{\beta}$ is n-dimensional. In our case we have $\boldsymbol{x} = [V_S, \tau]$, and $\boldsymbol{\beta} = [a, b, q, V_{C,c}, V_{C,s}, V_{C,t}, V_{C,0}]$. Given are m tuples of input and output, $(\boldsymbol{x}_i, y_i)$, where y_i is what *should* be the result of $f(\boldsymbol{x}_i, \boldsymbol{\beta})$. A least squares fitting method then tries to find a $\boldsymbol{\beta}'$ such that we get as close to the real function f as possible. Thus, it minimises the sum of squares

$$S(\boldsymbol{\beta}') = \sum_{i=1}^{m} (y_i - f(\boldsymbol{x}_i, \boldsymbol{\beta}'))^2. \tag{3.12}$$

3.4.1. Trust Region Reflective

The idea of the Trust Region Methods is explained clearly by Conn, Gold, and Toint (2000) [7] that the function that needs to be minimised — in our case $S(\boldsymbol{\beta})$ — can be approximated with $q(\boldsymbol{\beta})$, where q is a much simpler function. Let $\boldsymbol{\beta}_0$ be our first guess for $\boldsymbol{\beta}'$. We approximate S with q, but we only trust this approximation q on a neighbourhood of $\boldsymbol{\beta}_0$, which is called the Trust Region. By using this approximation of $q(\boldsymbol{\beta}_0)$, we can look for a $\boldsymbol{\beta}_1$ where $S(\boldsymbol{\beta}_1)$ is smaller than $S(\boldsymbol{\beta}_0)$.

We start with $\boldsymbol{\beta}_0$, and compute $S(\boldsymbol{\beta}_0)$. We make a model that we trust on a certain region, N_0, around $\boldsymbol{\beta}_0$, where we approximate S with q. With this model we can predict a point $\boldsymbol{\beta}^* \in N_0$ where

$$q(\boldsymbol{\beta}^*) < q(\boldsymbol{\beta}_0) \tag{3.13}$$

holds. We then compute $S(\boldsymbol{\beta}^*)$ and see if (3.13) also holds for S. If indeed we have

$$S(\boldsymbol{\beta}^*) < S(\boldsymbol{\beta}_0), \tag{3.14}$$

then we may confirm $\boldsymbol{\beta}_1 = \boldsymbol{\beta}^*$ and repeat the process. If (3.14) does not hold, then we trusted the approximation q too much, thus we reduce the size of N_0, and then predict a new point $\boldsymbol{\beta}^*$.

Now the question that remains is how we determine this q, and how exactly we find a point in N_0 that minimises q. This creates a subproblem. The Python SciPy library refers to Branch, Coleman, and Li (2000) [8], where they define $\psi_k(s)$ — the equivalent of our $q(\boldsymbol{\beta}_k)$ — as follows:

$$\psi_k(s) \stackrel{\text{def}}{=} s^T \nabla f_k + \frac{1}{2} s^T (\nabla^2 f_k + D_k \text{diag}(\nabla f_k) \mathbf{J}_k^v D_k) s, \tag{3.15}$$

where D_k is a scaling matrix. Then s, or in our case $\boldsymbol{\beta}^*$ can be found by the following minimisation subproblem:

$$\min_{s \in \mathbb{R}^n} \{ \psi_k(s) : \|D_k s\|_2 \leq \Delta_k \} \tag{3.16}$$

So now that we have the q and the means to find $\boldsymbol{\beta}^*$. We then keep finding new $\boldsymbol{\beta}_k$ until convergence is reached.

3.4.2. Levenberg-Marquardt

The Levenberg-Marquardt method is explained by Lourakis (2005) [9], and is only equipped to find a local minimum. It is an iterative process, and the user must provide an initial guess for $\boldsymbol{\beta}$, say $\boldsymbol{\beta}_0$. If there is only one minimum, any guess, like $\boldsymbol{\beta}_0 = (1, 1, ..., 1)^T$ is good. If, however, there are more local minima, then one must provide the method with a guess that is close to the global minimum.

We want to update this $\boldsymbol{\beta}_0$ to $\boldsymbol{\beta}_1 = \boldsymbol{\beta}_0 + \boldsymbol{\delta}$. We need to know which $\boldsymbol{\delta}$ to choose. For this, we first take the linearisation of f:

$$f(\boldsymbol{x}_i, \boldsymbol{\beta}_0 + \boldsymbol{\delta}) \approx f(\boldsymbol{x}_i, \boldsymbol{\beta}_0) + \mathbf{J}_i \boldsymbol{\delta}. \tag{3.17}$$

where $\mathbf{J}_i$ is the Jacobian of f with respect to $\boldsymbol{\beta}_0$

$$\mathbf{J}_i = \frac{\partial f(\boldsymbol{x}_i, \boldsymbol{\beta})}{\partial \boldsymbol{\beta}}. \tag{3.18}$$

Substituting (3.17) into (3.12) gives

$$S(\boldsymbol{\beta}_0 + \boldsymbol{\delta}) \approx \sum_{i=1}^{m} (y_i - f(\boldsymbol{x}_i, \boldsymbol{\beta}_0) - \mathbf{J}_i \boldsymbol{\delta})^2. \tag{3.19}$$

To be better able to solve this, we pose it in vector notation. For that, first note the following notation

$$\boldsymbol{y} = [y_1, ..., y_m]^T \tag{3.20}$$
$$\mathbf{J} = [\mathbf{J}_1, ..., \mathbf{J}_m]^T \tag{3.21}$$
$$\boldsymbol{f}(\boldsymbol{\beta}) = [f(\boldsymbol{x}_1, \boldsymbol{\beta}), ..., f(\boldsymbol{x}_m, \boldsymbol{\beta})]^T. \tag{3.22}$$

With this, we can write (3.19) as

$$\begin{aligned} S(\boldsymbol{\beta}_0 + \boldsymbol{\delta}) &\approx \left\| \boldsymbol{y} - \boldsymbol{f}(\boldsymbol{\beta}_0) - \mathbf{J}\boldsymbol{\delta} \right\|^2 \\ &= [\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{\beta}_0)]^T [\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{\beta}_0)] - 2[\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{\beta}_0)]^T \mathbf{J}\boldsymbol{\delta} + \boldsymbol{\delta}^T \mathbf{J}^T \mathbf{J}\boldsymbol{\delta}. \end{aligned} \tag{3.23}$$

Now, $S(\boldsymbol{\beta})$ has its minimum at a zero gradient with respect to $\boldsymbol{\beta}$, thus we take the derivative of $S(\boldsymbol{\beta}_0 + \boldsymbol{\delta})$ from (3.23), then set it to zero. This gives

$$(\mathbf{J}^T \mathbf{J})\boldsymbol{\delta} = \mathbf{J}^T [\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{\beta}_0)]. \tag{3.24}$$

This is where Levenberg comes in. He made a damped version of this equality, making his method an average between the Gauss-Newton algorithm and the gradient-descent. Using $\mathbf{I}$ as the identity matrix and λ as an adjustable constant, Levenberg says we should solve the following instead of (3.24):

$$(\mathbf{J}^T \mathbf{J} + \lambda \mathbf{I})\boldsymbol{\delta} = \mathbf{J}^T [\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{\beta}_0)]. \tag{3.25}$$

If the gradiant of S is high, we can keep the value of λ low to stay closer to the Gauss-Newton algorithm. If it does not decrease enough, we can use this addition of λ by giving it a higher value, thus getting closer to the gradient-descent direction.

One last addition from Fletcher [10] helps avoid slow convergence in the direction of small gradient. Instead of the identity matrix $\mathbf{I}$ we use $\mathrm{diag}(\mathbf{J}^T\mathbf{J})$.

Now that we have discovered the $\boldsymbol{\delta}$ needed to make $\boldsymbol{\beta}_1$ we start the process over with the new $\boldsymbol{\beta}_1$. We keep going, until either $\boldsymbol{\delta}$ or $S(\boldsymbol{\beta}_k) - S(\boldsymbol{\beta}_k + \boldsymbol{\delta})$ is sufficiently small. Then $\boldsymbol{\beta}_k$ is considered to be $\boldsymbol{\beta}'$ — the optimal solution.

3

4

Measurements

In this chapter we investigate the difference between the errors in the Iterative Method and the Direct Method, as well as the two different least squares fitting methods we explain in 3.4. We start with explaining what needs to be simulated, and then show the results of those simulations. After that, we will show the comparisons between the methods and the least squares methods in various situations.

We are given some existing model test data. These model tests are performed in a towing tank, without additional wave or wind resistance. We have six different simulations from this data. All six simulations have the same function of the current and the same non-dimensional time vector, namely:

$$\boldsymbol{\beta} = \begin{pmatrix} 1.5 \\ 1.3 \\ 0.8 \\ 1.5 \end{pmatrix}, \; \boldsymbol{\tau} = \begin{pmatrix} 0 \\ 0.07 \\ 0.12 \\ 0.18 \\ 0.26 \\ 0.33 \\ 0.38 \\ 0.47 \\ 0.52 \\ 0.62 \end{pmatrix} \tag{4.1}$$

As you can see from $\boldsymbol{\tau}$, there are 10 measurements in every simulation. The $(\boldsymbol{P}, V_G)$ tuples are listed in Tables 4.1-4.6.

4

	P_i (kW)	$V_{S,i}$ (kts)
1	5225	14
2	5225	14
3	6728	15
4	6728	15
5	8716	16
6	8716	16
7	11503	17
8	11503	17
9	11503	17
10	11503	17

Table 4.1: Data tuples of simulation 1.1

	P_i (kW)	$V_{S,i}$ (kts)
1	2313	11
2	3035	12
3	4018	13
4	5225	14
5	6728	15
6	8716	16
7	9987	16.5
8	9987	16.5
9	11503	17
10	11503	17

Table 4.2: Data tuples of simulation 1.2

	P_i (kW)	$V_{S,i}$ (kts)
1	34713	24
2	34713	24
3	40478	25
4	40478	25
5	47305	26
6	47305	26
7	55599	27
8	55599	27
9	65932	28
10	65932	28

Table 4.3: Data tuples of simulation 2.1

	P_i (kW)	$V_{S,i}$ (kts)
1	32141	23.5
2	34713	24
3	37483	24.5
4	40478	25
5	43737	25.5
6	47305	26
7	51238	26.5
8	55599	27
9	60466	27.5
10	65932	28

Table 4.4: Data tuples of simulation 2.2

	P_i (kW)	$V_{S,i}$ (kts)
1	1699	12
2	1699	12
3	2094	13
4	2094	13
5	2554	14
6	2554	14
7	3525	15
8	3525	15
9	5262	16
10	5262	16

Table 4.5: Data tuples of simulation 3.1

	P_i (kW)	$V_{S,i}$ (kts)
1	1483	11.5
2	1699	12
3	1896	12.5
4	2094	13
5	2296	13.5
6	2554	14
7	2939	14.5
8	3525	15
9	4304	15.5
10	5262	16

Table 4.6: Data tuples of simulation 3.2

A few notations and abbreviations which are used in this chapter:

Notation	Definition
MN	Measurement Noise
IT	Iterative Method, using Trust Region Reflective
IL	Iterative Method, using Levenberg-Marquardt
DT	Direct Method, using Trust Region Reflective
DL	Direct Method, using Levenberg-Marquardt
E_P	The error function of the Power $\sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(P(V_{S,i}) - P_i \right)^2}$
E_{V_C}	The error function of Current Speed $\sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(V_C(\tau_i) - V_{C,i} \right)^2}$
$E_{P,IT}$	E_P of IT
$E_{P,IL}$	E_P of IL
$E_{P,DT}$	E_P of DT
$E_{P,DL}$	E_P of DL
$E_{V_C,IT}$	E_{V_C} of IT
$E_{V_C,IL}$	E_{V_C} of IL
$E_{V_C,DT}$	E_{V_C} of DT
$E_{V_C,DL}$	E_{V_C} of DL

4.1. Example plots

To get an idea of the effectiveness of the methods, we use a visual result. In Figures 4.1-4.8, we show the result of fitting the data of simulation 1.2. Though differences between the two least squares fitting methods are not clear in these images, those between the Iterative Method and the Direct Method definitely are.

    Each image has three perspectives. For now, we plot just the actual data points. The inclusion of noise and inaccuracies in the measurements will be covered in Chapter 4.2. In addition to the data points, the approximation of the function is plotted. Every image shows either $V_C(\tau)$ or $P(V_S)$. The differences being the Method (Iterative or Direct) and the Fitter (Trust Region Reflective or Levenberg-Marquardt).
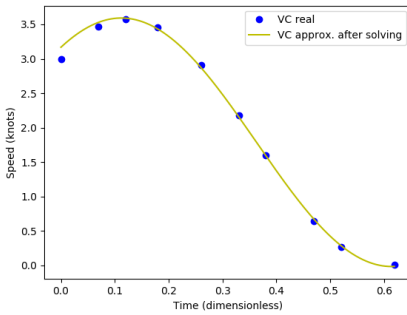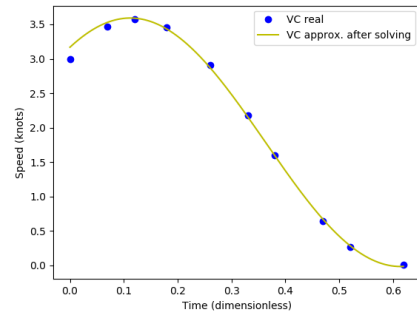

Figure 4.1: $V_C(\tau)$ with IT fitting.

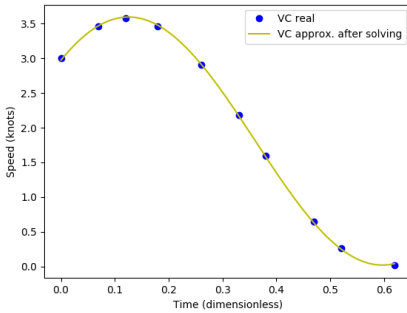Figure 4.2: $V_C(\tau)$ with IL fitting.

Figure 4.3: $V_C(\tau)$ with DT fitting.
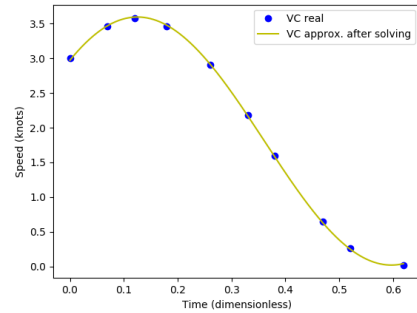
Figure 4.4: $V_C(\tau)$ with DL fitting.

We note that the $V_C(\tau)$ fitting of the Direct Method is clearly much better than that of the Iterative Method. A very slight difference can be seen in the $P(V_S)$ functions in figures 4.5-4.8, though it is not clear from the plot which one is better.

Figure 4.5: $P(V_S)$ with IT fitting.

Figure 4.6: $P(V_S)$ with IL fitting.
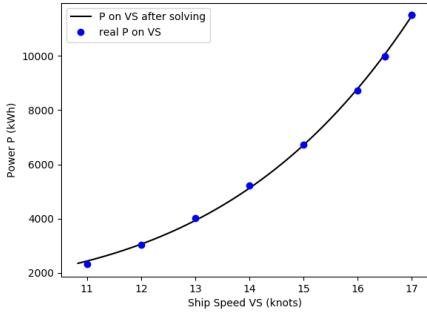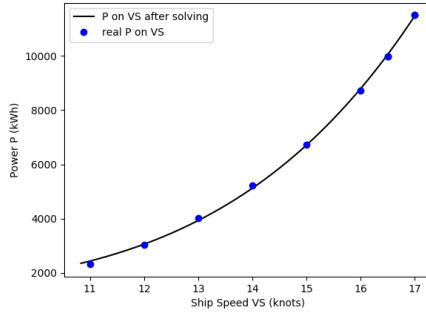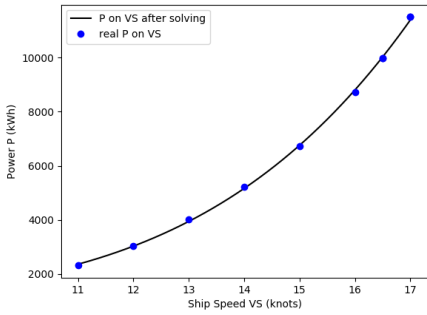

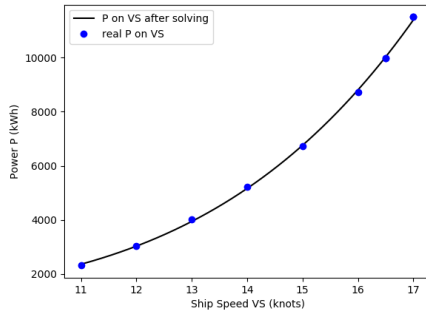Figure 4.7: $P(V_S)$ with DT fitting.

Figure 4.8: $P(V_S)$ with DL fitting.

4.2. Measurement Noise

As you see in the data, the measurements are exact. E.g., in Table 4.1, we see that the same power settings in subsequent rows correspond to exactly the same ship speed. Similarly, in the previous figures, we see that there are only "exact" data points, yet no "measured" data points. In reality, there will be noise in the measurements. Thus, we have to introduce this noise. It is more likely that the measurement is close to reality, and less likely that the measurement is far away from reality. A distribution that does this is the normal distribution.

We will add the noise to the real data points, thus we let the mean of the noise be 0. For the standard deviation, we need to have different values for P, V_G and t, as they are of different magnitudes. If we let the standard deviation be $\frac{1}{3}$ of the maximum error, 99.73% of the errors we create will be smaller than the maximum error. The other .27% are outliers, which can also happen in real life. The maximum errors are:

- P: according to Hans Huisman, the maximum error is desired to be 25 kW.

- V_G: the maximum error is 0.05 kts.

- t: the maximum error is 36 seconds. That means that the maximum error for τ is actually $\frac{36}{T_C}$.

When we include Measurement Noise, we run every simulation 1,000 times, and then take an average over the error values E_P and E_{V_C}. We do this, because the magnitude of the errors should depend on the magnitude of the Measurement Noise. The results are shown in Tables 4.7-4.12 below for the six different simulations.

In these results we can see that clearly the Measurement Noise makes some difference in the error that is produced. Michaël Mersie has noted that the biggest contributer to the error is the Measurement Noise in V_G.

One more notable thing about these results are Tables 4.11 and 4.12. Here we see that the Iterative Method has some problems. We will discuss this in more detail in Chapter 4.5.

	With MN	Without MN
$E_{P,IT}$	26.7065	15.6631
$E_{P,IL}$	27.1978	15.6631
$E_{P,DT}$	25.8026	15.3523
$E_{P,DL}$	25.1198	15.3523
$E_{V_C,IT}$	0.0113	0.0005
$E_{V_C,IL}$	0.0111	0.0005
$E_{V_C,DT}$	0.0117	0.0011
$E_{V_C,DL}$	0.0117	0.0011

Table 4.7: Different Errors with and without Measurement Noise, simulation 1.1

	With MN	Without MN
$E_{P,IT}$	92.1023	90.1089
$E_{P,IL}$	91.7498	90.1090
$E_{P,DT}$	58.0959	56.1560
$E_{P,DL}$	58.4677	56.1560
$E_{V_C,IT}$	0.0623	0.0606
$E_{V_C,IL}$	0.0617	0.0606
$E_{V_C,DT}$	0.0222	0.0180
$E_{V_C,DL}$	0.0225	0.0180

Table 4.8: Different Errors with and without Measurement Noise, simulation 1.2

	With MN	Without MN
$E_{P,IT}$	99.1648	72.9193
$E_{P,IL}$	95.5980	72.9196
$E_{P,DT}$	92.1038	68.3025
$E_{P,DL}$	91.0626	68.3025
$E_{V_C,IT}$	0.0116	0.0018
$E_{V_C,IL}$	0.0115	0.0018
$E_{V_C,DT}$	0.0118	0.0032
$E_{V_C,DL}$	0.0119	0.0032

Table 4.9: Different Errors with and without Measurement Noise, simulation 2.1

	With MN	Without MN
$E_{P,IT}$	98.6985	76.3623
$E_{P,IL}$	97.7673	75.5234
$E_{P,DT}$	87.9344	66.9800
$E_{P,DL}$	87.0323	66.9800
$E_{V_C,IT}$	0.0156	0.0096
$E_{V_C,IL}$	0.0152	0.0094
$E_{V_C,DT}$	0.0138	0.0078
$E_{V_C,DL}$	0.0139	0.0078

Table 4.10: Different Errors with and without Measurement Noise, simulation 2.2

	With MN	Without MN
$E_{P,IT}$	695.2947*	668.0375*
$E_{P,IL}$	N.A.	N.A.
$E_{P,DT}$	36.7763	36.2230
$E_{P,DL}$	36.9183	36.2230
$E_{V_C,IT}$	0.9495*	0.7073*
$E_{V_C,IL}$	N.A.	N.A.
$E_{V_C,DT}$	0.0334	0.0296
$E_{V_C,DL}$	0.0333	0.0296

Table 4.11: Different Errors with and without Measurement Noise, simulation 3.1.
Results marked with a * are only averaged over 10 simulations, as they took way too long.

	With MN	Without MN
$E_{P,IT}$	322.0079*	290.8738*
$E_{P,IL}$	N.A.	N.A.
$E_{P,DT}$	56.7570	56.3135
$E_{P,DL}$	56.3456	56.3135
$E_{V_C,IT}$	0.0119*	0.6709*
$E_{V_C,IL}$	N.A.	N.A.
$E_{V_C,DT}$	0.0690	0.0672
$E_{V_C,DL}$	0.0688	0.0672

Table 4.12: Different Errors with and without Measurement Noise, simulation 3.2.
Results marked with a * are only averaged over 10 simulations, as they took way too long. Results for IL are not strictly Levenberg-Marquardt, as that gives errors described in chapter 4.5

4.3. Finding situations where the Iterative Method beats the Direct Method

Firstly, let us look back at the experimental data in Tables 4.7-4.12. At first glance it seems that the Direct Method is better.

We see that in the error values of the P-function, the Direct Method performs better in all the simulations. Most margins are small (less than a factor of 2), but some are significant. However, in simulations 1.1 and 2.1, we notice that the Iterative Method produces a smaller error for the V_C-function. The margin here is approximately a factor of 2.

We now ask ourselves what simulations 1.1 and 2.1 have in common. Additionally, can we reproduce this or are these the only instances where this happens? For this we create theoretical data in Chapter 4.4. There, we first explain what the data in the tables means, and how it is made.

4.3.1. Strict Double Runs

Simulation data 1.1 and 2.1 both have strict Double Runs. Every second Run is strictly the same speed and power as the one before. We can easily reproduce this. For this, let us look at Tables 4.13, 4.15, and 4.17. These three simulations work with strict Double Runs. However, only Table 4.17 portrays what we are looking for: the Iterative Method performing better for V_C. Therefore, even if strictly Double Runs is a deciding factor, it is not the only one.

We note that 4.15 and 4.17 both have the same $\boldsymbol{\beta}$. Thus, this is not the deciding factor. We also note that 4.13 and 4.15 have the same values for a, b, q, and $\boldsymbol{V_S}$, while 4.17 has different values here. Thus, it is possible that the cause of a better performance in E_{V_C} from the Iterative Method is either the combination of a, b, and q, or the specifics of $\boldsymbol{V_S}$.

We note that Table 4.17 uses a lower value for a and a higher value for q than the other two Tables do. Additionally, the range of the speeds V_S is smaller in this Table than in the other two.

This latter hypothesis is backed up by the experimental data. The difference between the minimum and maximum V_S in simulation 1.1 and 2.1 is merely 3 and 4, respectively, while it is a little more for the other simulations.

4.3.2. Small range of V_S

To test this hypothesis, we make Table 4.21, where V_S ranges from 15 to 17.25, while it does not contain Double Runs. We also consider Table 4.20, where V_S ranges from 15 to 18, due to Double Runs.

In Table 4.21 we see that the Iterative Method performs badly compared to the Direct Method when using Trust Region Reflective, and actually fails when using Levenberg-Marquardt. This failure point is explained in Chapter 4.5.

In Table 4.20, however, we see that the Iterative Method outperforms the Direct Method in its approximation of $V_C(\tau)$. In addition to that, it even outperforms the Direct Method in its approximation for $P(V_S)$ when using Trust Region Reflective — and nearly so when using Levenberg-Marquardt.

This invalidates the theory that a small range for V_S would cause the Iterative Method to perform better than the Direct Method, but strengthens the theory that Double Runs do.

4.3.3. Less than enough values for the Direct Method

We take a look at one more way in which the Iterative Method can beat the Direct Method. Remember from Chapter 3 that the Iterative Process performs two distinct least-squares fits per iteration: once to find 3 unknown variables (a, b, and q), and once to find 4 unknown variables ($V_{C,c}$, $V_{C,s}$, $V_{C,t}$, and $V_{C,0}$). The Direct Method performs one single least-squares fit to find all 7 unknown variables.

The inherent problem that a set of equations with n unknown variables is that you need at least n equations to solve. Since Levenberg-Marquardt uses a linear solver in one of its steps, this problem will transfer. Therefore, if we have only 4 (P, V_S, τ)-tuples, the Direct Method does not have enough equations for the number of unknown variables. Thus, using Levenberg-Marquardt will not work there. Trust Region Reflective does accept less equations than its number of unknown values, though.

A way to circumvent this problem is to copy the *exact* data points we have, thereby doubling the (P, V_S, τ)-tuples we have. This extra data does not hold any new information, however, so we look at its effectiveness in Tables 4.22 and 4.23.

Firstly, we note that even though we expected the Iterative Method to succeed with only 4 data points (Table 4.22), it somehow converges to an extremely bad approximation. That said, we compare this Table to Table 4.23 and see that copying the data actually helps a lot. Although once in a while the Measurement Noise is just too weird for the Levenberg-Marquardt Method to converge, it seems that overall it should be possible to use this technique in case it is needed. Finally, the Direct Method actually still performs better than the Iterative Method.

4

4.4. Theoretical data

The following tables are the results of simulations with theoretical data — as opposed to experimental data. The data is created as follows:

- We take the values for V_S, then use our chosen a, b, and q to calculate the P.

- We use our chosen $\boldsymbol{\beta}$ as the constants in the function $V_C(\tau)$. With this we calculate V_C at all the measurement points.

- We make V_G from V_S and V_C.

- Noise is added to τ, P, and V_G. This is all that is provided to the solvers.

The resulting tables give the Errors produced by the solvers, differentiating between using the Iterative Method or the Direct Method, as well as whether we use Levenberg-Marquardt or Trust-Region Reflective.

Each table has different values for V_S, $\boldsymbol{\beta}$, and a, b, and q. These are all documented in the table captions.

	Iterative	Direct
$E_{P,T}$	8.2467	6.5992
$E_{P,L}$	8.5789	6.5363
$E_{V_C,T}$	0.0199	0.0183
$E_{V_C,L}$	0.0198	0.0179

Table 4.13: The average errors produced by both the Iterative and the Direct Method over 1000 simulations with Measurement Noise. Using the following data:
$a = 1050.5$; $b = 0.8$; $q = 2.9$.
$\boldsymbol{\beta} = (1.5, 1.3, 0.8, 1.5)$
$V_S = (11, 11, 15, 15, 19, 19, 23, 23, 27, 27)$.

	Iterative	Direct
$E_{P,T}$	9.4404	6.6977
$E_{P,L}$	9.3705	6.5552
$E_{V_C,T}$	0.0261	0.0189
$E_{V_C,L}$	0.0258	0.0187

Table 4.14: The average errors produced by both the Iterative and the Direct Method over 1000 simulations with Measurement Noise. Using the following data:
$a = 1050.5$; $b = 0.8$; $q = 2.9$.
$\boldsymbol{\beta} = (1.5, 1.3, 0.8, 1.5)$
$V_S =$
$(10, 12.5, 15, 17.5, 20, 20, 20, 20, 22.5, 25, 27.5)$.

	Iterative	Direct
$E_{P,T}$	8.4236	6.7674
$E_{P,L}$	8.2175	6.6064
$E_{V_C,T}$	0.0203	0.0184
$E_{V_C,L}$	0.0202	0.0184

Table 4.15: The average errors produced by both the Iterative and the Direct Method over 1000 simulations with Measurement Noise. Using the following data:
$a = 1050.5$; $b = 0.8$; $q = 2.9$.
$\boldsymbol{\beta} = (3.9, 0.9, 0.6, -0.2)$
$V_S = (11, 11, 15, 15, 19, 19, 23, 23, 27, 27)$.

	Iterative	Direct
$E_{P,T}$	9.6632	6.6621
$E_{P,L}$	9.6398	6.8921
$E_{V_C,T}$	0.0262	0.0195
$E_{V_C,L}$	0.0262	0.0187

Table 4.16: The average errors produced by both the Iterative and the Direct Method over 1000 simulations with Measurement Noise. Using the following data:
$a = 1050.5$; $b = 0.8$; $q = 2.9$.
$\boldsymbol{\beta} = (3.9, 0.9, 0.6, -0.2)$
$V_S =$
$(10, 12.5, 15, 17.5, 20, 20, 20, 20, 22.5, 25, 27.5)$.

	Iterative	Direct
$E_{P,T}$	33.3171	31.5166
$E_{P,L}$	33.2816	32.0342
$E_{V_C,T}$	0.0113	0.0119
$E_{V_C,L}$	0.0117	0.0117

Table 4.17: The average errors produced by both the Iterative and the Direct Method over 1000 simulations with Measurement Noise. Using the following data:
$a = 800$; $b = 0.08$; $q = 4.2$.
$\boldsymbol{\beta} = (3.9, 0.9, 0.6, -0.2)$
$\boldsymbol{V_S} = (17, 17, 18, 18, 18, 18, 19, 19, 20, 20)$.

	Iterative	Direct
$E_{P,T}$	7.8006	7.7742
$E_{P,L}$	7.7583	7.5855*
$E_{V_C,T}$	0.0167	0.0165
$E_{V_C,L}$	0.0163	0.0162*

Table 4.18: The average errors produced by both the Iterative and the Direct Method over 1000 simulations with Measurement Noise. Using the following data:
$a = 2500$; $b = 0.8$; $q = 2.9$.
$\boldsymbol{\beta} = (1.5, 1.3, 0.8, 1.5)$
$\boldsymbol{V_S} = (15, 15.5, 16, 16.5, 17, 17.5, 18, 18.5, 19, 19.5)$.
* The DL Method here diverges once every 100-400 times. These numbers represent the times that it converges.

	Iterative	Direct
$E_{P,T}$	10.3475**	7.4966
$E_{P,L}$	N.A.	7.3785*
$E_{V_C,T}$	0.0187**	0.0188
$E_{V_C,L}$	N.A.	0.0190*

Table 4.19: The average errors produced by both the Iterative and the Direct Method over 1000 simulations with Measurement Noise. Using the following data:
$a = 2500$; $b = 0.8$; $q = 2.9$.
$\boldsymbol{\beta} = (1.5, 1.3, 0.8, 1.5)$
$\boldsymbol{V_S} = (15, 15, 15, 15.5, 15.5, 15.5, 16, 16, 16, 16.5)$.
* The DL Method here diverges once every 100-400 times. These numbers represent the times that it converges.
** The IL Method here diverges over half the time. The number here, therefore, represents the average of only 5 simulations where it did converge.

	Iterative	Direct
$E_{P,T}$	8.5049	8.7105
$E_{P,L}$	8.4218	8.3927*
$E_{V_C,T}$	0.0199	0.0202
$E_{V_C,L}$	0.0198	0.0200*

Table 4.20: The average errors produced by both the Iterative and the Direct Method over 1000 simulations with Measurement Noise. Using the following data:
$a = 2500$; $b = 0.8$; $q = 2.9$.
$\boldsymbol{\beta} = (1.5, 1.3, 0.8, 1.5)$
$\boldsymbol{V_S} = (15, 15, 16, 16, 17, 17, 18, 18)$.
* The DL Method here diverges once every 10-400 times. These numbers represent the times that it converges.

	Iterative	Direct
$E_{P,T}$	8.9136*	6.9738
$E_{P,L}$	N.A.**	7.0785
$E_{V_C,T}$	0.0208*	0.0172
$E_{V_C,L}$	N.A.**	0.0175

Table 4.21: The average errors produced by both the Iterative and the Direct Method over 1000 simulations with Measurement Noise. Using the following data:
$a = 2500$; $b = 0.8$; $q = 2.9$.
$\boldsymbol{\beta} = (1.5, 1.3, 0.8, 1.5)$
$\boldsymbol{V_S} = (15, 15.25, 15.5, 15.75, 16, 16.25, 16.5, 16.75, 17, 17.25)$.
* This number is averaged over only 9 simulations, as they took way too long, and one diverged.
** The IL Method here diverges every time, and thus does not produce results. See Chapter 4.5

	Iterative	Direct
$E_{P,T}$	9230.2713	111.9914
$E_{P,L}$	9230.2713	N.A.*
$E_{V_C,T}$	4.9987	0.1889
$E_{V_C,L}$	5.0010	N.A.*

Table 4.22: The average errors produced by both the Iterative and the Direct Method over 1000 simulations with Measurement Noise. Using the following data:
$a = 2500;\ b = 0.8;\ q = 2.9.$
$\boldsymbol{\beta} = (1.5, 1.3, 0.8, 1.5)$
$\boldsymbol{V_S} = (16, 18, 20, 22).$
$\boldsymbol{\tau} = (0, 0.07, 0.12, 0.18).$
* The DL Method here does not have enough values to solve, thus produces no result.

	Iterative	Direct
$E_{P,T}$	289.3705	72.2452
$E_{P,L}$	288.9065**	111.0208*
$E_{V_C,T}$	0.4275	0.1151
$E_{V_C,L}$	0.4282**	3.8168*

Table 4.23: The average errors produced by both the Iterative and the Direct Method over 1000 simulations with Measurement Noise. Using the following data:
$a = 2500;\ b = 0.8;\ q = 2.9.$
$\boldsymbol{\beta} = (1.5, 1.3, 0.8, 1.5)$
$\boldsymbol{V_S} = (16, 18, 20, 22, 16, 18, 20, 22).$
$\boldsymbol{\tau} = (0, 0.07, 0.12, 0.18, 0, 0.07, 0.12, 0.18).$
* The DL Method here diverges once every 50-300 times. These numbers represent the times that it converges.
** The IL Method here diverges once every 1-100 times. These numbers represent the average of 500 simulations where it did not diverge.

4.5. Major problem with the Iterative Method for low Power Settings

Reconsidering Figure 3.1, we focus on the first step after checking the stop condition. In this step, we set intermediate values for the ship's speed, $V'_{S,i}$, using the given $P_{id,i}$ and the latest values for a, b, and q.

At first glance, it seems like the logical way to calculate V_S — simply the inverse function of $P(V_S)$, namely $V_S = \sqrt[q]{\frac{P_{id,i}-a}{b}}$. However, since the Iterative Method tries to improve the values for a, b, and q step by step, it is possible that we quickly get to an iteration where the value for a can exceed the lowest value of $P_{id,i}$. This leads to taking a root of a negative number, which does not produce real values. As the output has to be a speed, we do expect a real value.

Thus, we need a way to avoid the value of a exceeding $\min_i(P_{id,i})$. One way is to set an upper bound on the values that can be guessed. Trust Region Reflective supports this, while Levenberg-Marquardt do not support bounds. Therefore, as soon as a will exceed $\min_i(P_{id,i})$, we should switch over to Trust Region Reflective.

However, when trying to implement this, q will sometimes get bigger than 3000. This means that we would also have to set a bound on q. All this hints at the fact that the Iterative Method does not reliably converge. The only way to make it converge is to set bounds that conform to the expectations we have from the physics. These bounds lead to an Error that can get over 20 times bigger than the Error given by the Direct Method. See Table 4.24.

$E_{P,IT}$	788.7327
$E_{P,DT}$	37.5844
$E_{V_C,IT}$	1.0328
$E_{V_C,DT}$	0.0382

Table 4.24: Showing the big error in low Power settings for the Iterative Method. Simulation 3.1. The constants in the V_C function are $(V_{C,c}, V_{C,s}, V_{C,t}, V_{C,0})$ = (3.9, 0.9, 0.6, -0.2).

This can also be seen in figures 4.9 and 4.11. To compare, see the fit from the Direct Method of the same simulation in figures 4.10 and 4.12. Before we analyze it, let us first describe the figures.

In Figure 4.9 we see a plot of the delivered power, as a function of the ship's speed. The blue dots represent the real data points. The red crosses represent the data points with Measurement Noise added to it. There we also see how small this Measurement Noise really is. Lastly, the black line is $P(V_S)$ with a, b, and q as found by the solver (Iterative Method, in this case).

Next, in Figure 4.11 we see multiple plots. All of them plot speed as a function of time. The blue dots represent the real speed of the current over time, where the yellow line is the approximation of this V_C-function, as found by the solver. The red crosses represent the ship's speed relative to the ground, with Measurement Noise. The reason that it zigzags is because every other Run, the ship sails with or against
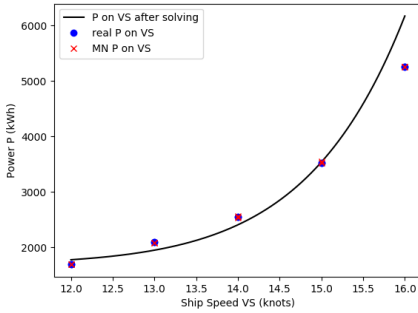
Figure 4.9: A visual representation of $P(V_S)$ with the found constants by the Iterative Method from simulation 3.1.
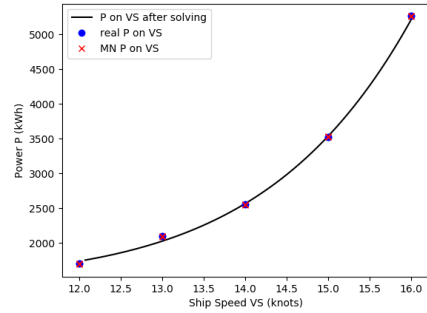

Figure 4.10: A visual representation of $P(V_S)$ with the found constants by the Direct Method from simulation 3.1.
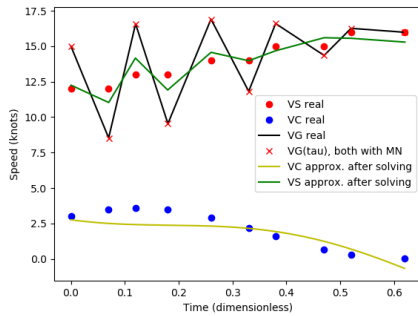

Figure 4.11: A visual representation of $V_S(\tau)$, $V_G(\tau)$, and $V_C(\tau)$ with the found constants by the Iterative Method from simulation 3.1.
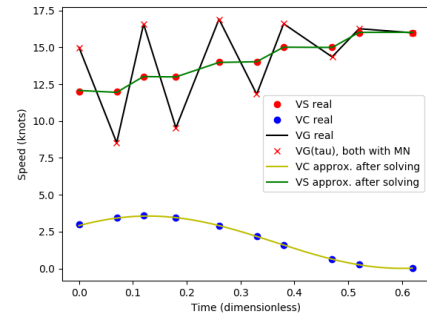

Figure 4.12: A visual representation of $V_S(\tau)$, $V_G(\tau)$, and $V_C(\tau)$ with the found constants by the Direct Method from simulation 3.1.

the current. This means that when the current increases the ship's speed in one Run, it decreases it in the next. Lastly, the red dots are the real values of the ship's speed, V_S. The solver approximates this by adding V_G and V_C appropriately. That is why the green line, which is the approximation, is close when its approximation for V_C is close, and wrong when its approximation for V_C is wrong.

This illustrates the downside of the Iterative Method. In the good cases, bouncing back and forth between adjusting the constants in either the P-function or the V_S-function improves both, little by little. In the bad cases like this one, however, a bad adjustment to one will make the adjustment for the other even worse. We see in Figure 4.11 that the V_C function is fitted quite horribly. This, in turn, ruins the calculated V_S. The effect of this is seen in Figure 4.9, where the values of V_S that are used are too low to accurately estimate the constants in the P-function. We might say the calculations diverge.

This emphasises the effectiveness of the Direct Method. As there are no more

loose steps, but only one coherent Least Squares problem over the entire system of equations, the calculations do not influence other calculations to diverge. We see this in the fact that where the Iterative Method crashes with an error, the Direct Method gives an answer in little time.

4

4.6. Effectiveness of one step of Mean of Means

In this section we evaluate how effective it is for the Iterative Method to start off with one step of Mean of Means before the first iteration. We can divide the values for V_G up in pairs of subsequent values, then set V_S of both values in the pair to the average of the two. Visually, this means that we do the following

instead of setting $V'_{S,i}$ according to 3.1, in the first iteration. In Tables 4.25-4.30 we see the effectiveness this has on the Simulation data.

Note that the Iterative Method did not have problems with Simulation data 1.1, 1.2, 2.1, and 2.2. The first four Tables therefore give us insight into whether an instance of Mean of Means increases efficiency of the Iterative Method. We see that this is most definitely the case. Although we are talking about milliseconds per simulation, the instance of Mean of Means greatly decreases the time it takes to complete the Process - even to less than a third of the time.

This alone, however, is not enough reason to implement this. For every ship that does a speed trial - and takes a few hours to do this - we save a few milliseconds in approximating its $P(V_S)$-function. However, it can be noted that - though the margin is small - the Iterative Method also benefits from an instance of Mean of Means by improving or approximately equaling all its errors. Well, all errors except $E_{P,T}$ for Simulation data 2.1.

So, the evidence seems to suggest that if the Iterative Process does converge, then it converges faster when an instance of Mean of Means is used before the first iteration.

In the last two Tables, we observe the effect on diverging Iterative Processes. For Simulation data 3.1 in Table 4.29 we see that Trust Region Reflective produced enormous errors, and Levenberg-Marquardt does not even produce any values. Neither of these problems are solved by introducing an instance of Mean of Means to the process. For Simulation data 3.2 in Table 4.30 we observe the same thing.

	MoM	No MoM
$E_{P,T}$	22.7002	22.7538
$E_{P,L}$	22.8320	22.8807
$E_{V_C,T}$	0.0112	0.0113
$E_{V_C,L}$	0.0111	0.0113
Average loops T	8.044	9.363
Average loops L	8.056	9.397
Time T*	296.2	345.6
Time L*	54.5	68.9

Table 4.25: The effect of performing one iteration of Mean of Means before starting the Iterative Process. Simulation data 1.1.
* Average time per simulation in milliseconds.

	MoM	No MoM
$E_{P,T}$	76.3366	76.8503
$E_{P,L}$	76.1294	76.3996
$E_{V_C,T}$	0.0627	0.0635
$E_{V_C,L}$	0.0619	0.0629
Average loops T	20.120	22.465
Average loops L	20.102	22.372
Time T*	244.5	299.8
Time L*	81.8	96.2

Table 4.26: The effect of performing one iteration of Mean of Means before starting the Iterative Process. Simulation data 1.2.
* Average time per simulation in milliseconds.

	MoM	No MoM
$E_{P,T}$	114.0469	112.8976
$E_{P,L}$	112.9890	115.6962
$E_{V_C,T}$	0.0116	0.0118
$E_{V_C,L}$	0.0115	0.0116
Average loops T	13.830	18.980
Average loops L	13.583	19.026
Time T*	577.2	1862.7
Time L*	105.8	296.1

Table 4.27: The effect of performing one iteration of Mean of Means before starting the Iterative Process. Simulation data 2.1.
* Average time per simulation in milliseconds.

	MoM	No MoM
$E_{P,T}$	129.2512	131.2432
$E_{P,L}$	127.7448	129.3305
$E_{V_C,T}$	0.0156	0.0157
$E_{V_C,L}$	0.0151	0.0152
Average loops T	20.318	25.819
Average loops L	19.994	26.139
Time T*	760.0	2337.0
Time L*	138.2	354.9

Table 4.28: The effect of performing one iteration of Mean of Means before starting the Iterative Process. Simulation data 2.2.
* Average time per simulation in milliseconds.

	MoM	No MoM
$E_{P,T}$	381.7553	378.7258
$E_{P,L}$	N.A.**	N.A.**
$E_{V_C,T}$	0.9371	0.9656
$E_{V_C,L}$	N.A.**	N.A.**
Average loops T	49.980	59.22
Average loops L	N.A.**	N.A.**
Time T*	5771.0	6714.0
Time L*	N.A.**	N.A.**

Table 4.29: The effect of performing one iteration of Mean of Means before starting the Iterative Process. Simulation data 3.1.
* Average time per simulation in milliseconds.
** The IL Method here diverges every time, and thus does not produce results.

	MoM	No MoM
$E_{P,T}$	396.7838	324.4158
$E_{P,L}$	N.A.	N.A.
$E_{V_C,T}$	1.1161	0.5603
$E_{V_C,L}$	N.A.	N.A.
Average loops T	366.9***	384.4**
Average loops L	N.A.	N.A.
Time T*	34130***	35340**
Time L*	N.A.	N.A .

Table 4.30: The effect of performing one iteration of Mean of Means before starting the Iterative Process. Simulation data 3.2. The shown averages are only from 10 simulations, as they took too long.
* Average time per simulation in milliseconds.
** The time cap for 35 seconds per simulation had been reached every time (*** except for one time). This happens when the Iterative Method is diverging.

5

Conclusion

To conclude, we review everything we have found in this dissertation.

First, we found that the Direct Method in most cases outperforms the Iterative Method. However, there are some specific cases where the Iterative Method is better in some areas than the Direct Method. We notice that when the speed trial consists only of Double Runs, and the difference between the minimum V_S and maximum V_S is small, that the Iterative Method will likely have a better approximation of $V_C(\tau)$ than the Direct Method has. In one of those cases, it also had a better approximation of $P(V_S)$.

The question must therefore be asked, which of these is more important? A better approximation of $V_C(\tau)$, or a better approximation of $P(V_S)$. The answer is left to the user to decide. Note, however, that unless the aforementioned conditions are met, it is likely that the Direct Method will give a better approximation for both $V_C(\tau)$ and $P(V_S)$ than the Iterative Method will.

We even see that the Direct Method's problem of needing more data points can be circumvented by just duplicating the data you already have, and it will still outperform the Iterative Method. Why this happens is not yet clear, thus it might be useful to study this more in later research. It can still be discovered in which situations it does work, and in which it does not.

From our research it follows that the reason for the Iterative Method's unstableness does not apply to the Direct Method. Since the Direct Method optimises all unknown constants at the same time, instead of in two groups, it eliminates a path of divergence that the Iterative Method suffers from.

Lastly we observed that, for the Iterative Method, there is a clear advantage of performing an instance of Mean of Means in the first iteration. If the Iterative Method would already converge, the instance of Mean of Means reduces the number of iterations needed. This, in turn, reduces the time the whole process takes. And the error values do not suffer for it.

If, however, the Iterative Method diverges, it may still do so when performing an instance of Mean of Means.

In our data we have slightly touched upon the differences of using Levenberg-Marquardt or Trust Region Reflective. It seems that for the Direct Method, in general the errors are smaller when using Levenberg-Marquardt. However, it is also more prone to diverging, while Trust Region Reflective is quite stable. It could be interesting for a follow-up research to find out why this might be the case.

References

[1] F. Buwalda, *Analysis of methods for determining ship speed during a sea trial*, BSc Thesis (TU Delft, http://resolver.tudelft.nl/uuid: e7d5f3b6-324e-42ae-af94-7be1b5a0a83f, 2016).

[2] M. Mersie, *Determination of water speeds of ships*, BSc Thesis (TU Delft, http://resolver.tudelft.nl/uuid: b9ff7017-647e-47f4-88c5-189d2c433268, 2017).

[3] J. D. van Manen and P. van Oossanen, *Principles of Naval Architecture, Volume II: Resistance, Propulsion and Vibration*, edited by E. V. Lewis (The Society of Naval Architects and Marine Engineers, 1988).

[4] E. C. Tupper, *Introduction to Naval Architecture* (Butterworth Heinemann, 1996).

[5] *ISO15016: Ships and marine technology — Guidelines for the assessment of speed and power performance by analysis of speed trial data* (ISO and ITTC, 2015).

[6] G. Strasser, K. Takagi, S. Werner, U. Hollenbach, T. Tanaka, K. Yamamoto, and K. Hirota, *A verification of the ITTC/ISO speed/power trials analysis,* Journal of Marine Science and Technology **20** (2015), 10.1007/s00773-015-0304-7.

[7] A. R. Conn, N. I. M. Gould, and P. L. Toint, *Trust-Region Methods* (Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000).

[8] M. A. Branch, T. F. Coleman, and Y. li, *A subspace, interior, and conjugate gradient method for large-scale bound-constrained minimization problems,* SIAM Journal on Scientific Computing **21** (1999), 10.1137/S1064827595289108.

[9] M. I. Lourakis *et al.*, *A brief description of the levenberg-marquardt algorithm implemented by levmar,* Foundation of Research and Technology **4**, 1 (2005).

[10] R. Fletcher and United Kingdom Atomic Energy Authority, *Modified Marquardt Subroutine for Non-Linear Least Squares.*, Tech. Rep. (Theoretical Physics Division, Atomic Energy Research Establishment, 1971).

A
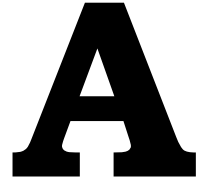
Python Code

The following are the 4 different python files that were used to obtain the data in this dissertation. The files can also be downloaded directly from Google Drive at http://bit.ly/MAtsmaBEPPython for ease.

A

A.1. Main

```python
from Formulas import *
from PracData import *
from TheoData import *
import matplotlib as mpl
import matplotlib.pyplot as plt

### Global constants. These are the only things that need changing
### in order to get different kinds of results.
simulations = 1000
timeLimit = 3500 # time limit in seconds of all
                 # simulations combined using Iterative Method
chosen_simulation_data = 1 # 1-6 is Practical data;
                           # 7+ is Theoretical data
MoM = 0 # Whether a Mean of Means should be
        # done on VG for the Iterative Method
first_guess_for_abq = [1000, 2, 3]
first_guess_for_vcConstants = [1., 1., 1., 1.]
vcCorrect = [1.5, 1.3, 0.8, 1.5]
method = 0 # 0 for Levenberg-Marquadt, 1 for Trust-Region Reflective
canBound = 0 # Whether or not changing to TRR with bounds is allowed
DIRECTorITERATIVE = 0 # 0 for direct, 1 for iterative
NoiseOnP = 1       # To set Measurement Noise
NoiseOnVG = 1      # on the different variables.
NoiseOnTAU = 1     # 0 for no noise, 1 for noise.

### Don't change
timeLimit /= float(simulations)
method = ['lm', 'trf'][method]
DIRECTorITERATIVE = ['direct', 'iterative'][DIRECTorITERATIVE]

sim_numbers = [1.1, 1.2, 2.1, 2.2, 3.1, 3.2]
sim_numbers += ["Theoretical situation " + str(i)
                for i in range(1,20)]
sim_name = sim_numbers[chosen_simulation_data-1]

if chosen_simulation_data <= 6:
    allP = pracP
    allV = pracV
    tau = pracTau
else:
    allP = theoP
    allV = theoVS
    chosen_simulation_data -= 6
    tau = theoTau[chosen_simulation_data-1]
```

```python
### Main
if __name__=='__main__':
    print "Starting main."

    ## Solve sample data
    # Real data. User doesn't get this.
    P = allP[chosen_simulation_data-1]
    VS = allV[chosen_simulation_data-1]
    VC = TAUtoVC(tau, *vcCorrect)
    sLeftRight = np.array([(-1)**i for i in range(len(VS))])
    VG = VSandTAUandVCCONSTtoVG(VS, sLeftRight, tau, vcCorrect)

    # Do 1000 simulations with this data
    startTime = time.time()
    results = [[None]*7]*simulations
    prints = 1
    for simulation in range(simulations):
        # Some helpful prints for myself
        simStartTime = time.time()
        if ((simulations < 10) or
            (simulation%(simulations/10) == 0) or
            ((simStartTime-startTime)/40 > prints)):
            if simulation != 0:
                print "Done", simulation, "simulations in",
                print round(simStartTime-startTime, 1), "seconds."
                prints += 1

        # Adding measurement noise. This is what the user gets.
        VG_MN, P_MN, tau_MN = MakeNoise(VG, NoiseOnVG, P,
                                        NoiseOnP, tau, NoiseOnTAU)

        # First guess
        abq0 = [el for el in first_guess_for_abq]
        vcConst0 = [el for el in first_guess_for_vcConstants]

        # Find best values
        if DIRECTorITERATIVE=='iterative':
            try:
                res = IterativeMethod(abq0, vcConst0, P_MN, VG_MN,
                                      tau_MN, method=method, MoM=MoM,
                                      epsilon=10e-3,
                                      timeLimit=timeLimit,
                                      canBound=canBound)
            except:
```

```python
                print "\nLast sim was", simulation
                results = results[:simulation-1]
                break
        elif DIRECTorITERATIVE=='direct':
            try:
                res = DirectMethod(abq0, vcConst0, P_MN, VG_MN,
                                   tau_MN, method=method)
            except RuntimeWarning:
                print "\nLast sim was", simulation
                results = results[:simulation-1]
                break

        abq1, vcConst1, EP, loops = res

        # Calculating the real error on P function
        CalcP = VStoP(VS, *abq1)
        EPReal = Error(CalcP, P)

        # Calculating error on VC function
        VC_BasedOnConstants = TAUtoVC(tau, *vcConst1)
        EVC = Error(VC_BasedOnConstants, VC)

        # Store best values
        simDuration = time.time()-simStartTime
        results[simulation] = [abq1, vcConst1, EP, EVC,
                               loops, simDuration, EPReal]

print "Done all", simulations, "simulations in",
print round(time.time()-startTime, 1), "seconds."

# Processing results
resultsarr = np.array(results)
abqAll = resultsarr[:, 0]
vcConstAll = resultsarr[:, 1]
EPAll = resultsarr[:, 2]
EPaverage = EPAll.sum()/len(EPAll)
EVCAll = resultsarr[:, 3]
EVCaverage = EVCAll.sum()/len(EVCAll)
loopsAll = resultsarr[:, 4]
loopsAverage = float(loopsAll.sum())/len(loopsAll)
simDurationAll = resultsarr[:, 5]
totalDuration = time.time()-startTime
EPRealAll = resultsarr[:, 6]
EPRealAverage = EPRealAll.sum()/len(EPRealAll)
```

```python
# Only non-diverging results
EPconv = []
EPRconv = []
EVCconv = []
for i in range(len(EPAll)):
    if EPAll[i]<30 and EPRealAll[i]<30 and EVCAll[i]<0.05:
    # 30 and 0.05 are chosen arbitrarily;
    # they should differ per dataset, and here merely
    # offer just some insight into the following data
        EPconv.append(EPAll[i])
        EPRconv.append(EPRealAll[i])
        EVCconv.append(EVCAll[i])
if len(EPconv)>0:
    EPconv = np.array(EPconv)
    EPRconv = np.array(EPRconv)
    EVCconv = np.array(EVCconv)
    EPcAv = EPconv.sum()/len(EPconv)
    EPRcAv = EPRconv.sum()/len(EPRconv)
    EVCcAv = EVCconv.sum()/len(EVCconv)
else:
    EPcAv = "None"
    EPRcAv = "None"
    EVCcAv = "None"


# Printing useful stuff

print "\nSimulation was", sim_name
print "vcConstants were", vcCorrect
print "VS was", VS
print "P was", P
print "NoiseOnP", NoiseOnP
print "NoiseOnVG", NoiseOnVG
print "NoiseOnTAU", NoiseOnTAU
print "Method was", DIRECTorITERATIVE
print "Least squares was", method
if DIRECTorITERATIVE=='iterative':
    print "MoM", MoM
print "Self-reported average EP is", EPaverage
print "Average EVC is", EVCaverage
print "Real average EP is", EPRealAverage
print "Average number of loops was", loopsAverage

print "\nEPcAv, EPRcAv, EVCcAv:"
print EPcAv, EPRcAv, EVCcAv
```

A

```python
# Example plot
taus = np.arange(tau_MN[0], tau_MN[-1], .001)
VCs = TAUtoVC(taus, *vcConst1)
VC1 = TAUtoVC(tau_MN, *vcConst1)
VS1 = VGandLRandVCtoVS(VG_MN, sLeftRight, VC1)
VSs = np.arange(VS1[0], VS1[-1], .001)
Ps = VStoP(VSs, *abq1)

plt.figure(1)
plt.plot(tau, VS, 'ro', label="VS real")
plt.plot(tau, VC, 'bo', label="VC real")
plt.plot(tau, VG, 'k', label="VG real")

if NoiseOnVG and NoiseOnTAU:
    plt.plot(tau_MN, VG_MN, 'rx',
             label="VG(tau), both with MN")
elif NoiseOnVG and not NoiseOnTAU:
    plt.plot(tau_MN, VG_MN, 'rx',
             label="VG(tau), with MN on VG")
elif not NoiseOnVG and NoiseOnTAU:
    plt.plot(tau_MN, VG_MN, 'rx',
             label="VG(tau), with MN on tau")

plt.plot(taus, VCs, 'y', label="VC approx. after solving")
plt.plot(tau_MN, VS1, 'g', label="VS approx. after solving")

plt.legend()
plt.xlabel("Time (dimensionless)")
plt.ylabel("Speed (knots)")

plt.figure(2)
plt.plot(VSs, Ps, 'k', label="P on VS after solving")
plt.plot(VS, P, 'bo', label="real P on VS")
if NoiseOnP:
    plt.plot(VS, P_MN, 'rx', label="MN P on VS")

plt.legend()
plt.xlabel("Ship Speed VS (knots)")
plt.ylabel("Power P (kWh)")

plt.show()
```

A.2. Formulas

```python
import time
import numpy as np
from scipy.optimize import least_squares as ls
import warnings
warnings.filterwarnings('error')

### Global constants
T_C = 12*3600 + 25*60 + 12 # Time of a tide in seconds
pi = np.pi
small = 10**(-4)
timeLimit = 2000/10 # Time limit of one whole Iterative
                    # Method in seconds

### Functions
cos = np.cos
sin = np.sin

def residual(beta, function, input_value, output_value):
    """This function takes an estimate (beta) for the constants
in a function (function), and returns the difference between
the given output_values and the calculated output based on the
input_values and the aforementioned beta and function."""
    return output_value - function(input_value, *beta)

def VStoP(V_S, a, b, q):
    """We have established that P(V_S)=a+b*(V_S)^q
This function takes V_S, a, b, q, and returns P.
Also accepts array for V_S."""
    try:
        return a+b*(V_S**q)
    except:
        print "\nWe have an error with VStoP"
        print "a:", a
        print "b:", b
        print "q:", q
        print "VS:", V_S
        return a+b*(V_S**q)

def PtoVS(P, a, b, q):
    """Same as VStoP but the other way around.
Also accepts array for P."""
    try:
        return ((P-a)/float(b))**(1./q)
    except Warning:
```

```python
        print "\n\n"+"-"*30 + "WE HAVE A WARNING!!" + "-"*30+"\n\n"
        print P, a, b, q


def TtoTAU(t):
    """Takes the time in seconds (t), and returns the
dimensionless value for time (tau), based on the global T_C.
Also accepts array for t."""
    return t/float(T_C)


def TAUtoVC(tau, VCcos, VCsin, VCt, VC0):
    """When given the dimensionless value for time (tau),
this function returns the speed of the current (V_C).
Also need the constants VCcos, VCsin, VCt, VC0.
Also accepts array for tau."""
    return (VCcos*cos(2*pi*tau) + VCsin*sin(2*pi*tau)
            + VCt*tau + VC0)


def VGandVSandLRtoAVCandLR(VG, VS, sLeftRight):
    """This function takes the Ground Speed (VG), the Ship Speed
(VS) and the direction the ship is sailing in (sLeftRight,
either -1 or +1). Then returns a tuple with the absolute
value of the Speed of the Current and the direction of
the current (either -1 or +1)."""
    VC = VG-VS
    if VC<0:
        # This means the ship's speed is greater than its measured
        # ground speed. Therefore, the current is working against
        # us in this Run. Thus, the direction of the current is
        # the opposite of that of the ship.
        cLeftRight = -sLeftRight
        VC = -VC # for absolute value.
    else:
        # This means the ship's speed is less than its measured
        # ground speed. Therefore, the current is working with
        # us in this Run. Thus, the direction of the current is
        # is the same as that of the ship.
        cLeftRight = sLeftRight
    return VC, cLeftRight


def VGandVSandLRtoAVCandLRarr(VGarr, VSarr, sLeftRightarr):
    """Same as VGandVStoAVCandLR but for arrays as inputs.
Also accepts singular numbers instead of arrays."""
    VCarr = VGarr-VSarr
    cLeftRightarr = sLeftRightarr*np.sign(VCarr)
    return abs(VCarr), cLeftRightarr
```

```python
def AVCandLRtoVC(AVC, cLeftRight):
    """Takes the absolute value of VC (AVC), and whether VC
is going left or right (cLeftRight, -1 or +1), then
returns VC to the right. So if VC is going left, this
returns negative VC. Also accepts arrays as inputs."""
    return AVC*cLeftRight


def VGandVSandLRtoVC(VG, VS, sLeftRight):
    """Takes VG and VS and whether the ship goes right or
left (-1 or +1), then returns VC.
Accepts both arrays or singular numbers."""
    if type(VG) == np.ndarray:
        AVC, cLeftRight = VGandVSandLRtoAVCandLRarr(VG, VS,
                                             sLeftRight)
    else:
        AVC, cLeftRight = VGandVSandLRtoAVCandLR(VG, VS,
                                             sLeftRight)
    return AVCandLRtoVC(AVC, cLeftRight)


def VCtoAVCandLR(VC):
    """Takes VC and returns a tuple with its absolute value
and whether it is going left or right (-1 or +1).
ONLY ACCEPTS ONE NUMBER (no array)."""
    if np.sign(VC) != 0:
        return abs(VC), np.sign(VC)
    else:
        return 0., 1.


def VCtoAVCandLRarr(VC):
    """Same as VCtoAVCandLR, but for VC is an array.
ONLY ACCEPTS ARRAY."""
    vals = np.array(VC)
    signs = np.sign(VC)
    vals = vals*signs
    for i in range(len(signs)):
        if signs[i]==0:
            signs[i]==1.
    return vals, signs


def VGandLRandAVCandLRtoVS(VG, sLeftRight, AVC, cLeftRight):
    """Takes VG and AVC and the directions of the ship and the
current (-1 or +1), then returns VS.
Also accepts arrays as inputs."""
    # If the ship and the current go in the same direction,
```

```python
    # then AVC should be subtracted from VG to get VS.
    # In this case, sLR and cLR have the same sign, thus
    # multiply to +1.
    #
    # If the ship goes against the current,
    # then AVC should be added to VG to get VS.
    # In this case, sLR and cLR have opposite signs, thus
    # multiply to -1.
    #
    # Therefore, we can always subtract sLR*cLR the value of AVC
    return VG - sLeftRight*cLeftRight*AVC

def VGandLRandVCtoVS(VG, sLeftRight, VC):
    """Takes VG and whether the ship goes left or rigpht, and
VC, then returns VS. Also accepts arrays as inputs."""
    if type(VC) == np.ndarray:
        AVC, cLeftRight = VCtoAVCandLRarr(VC)
    else:
        AVC, cLeftRight = VCtoAVCandLR(VC)
    return VGandLRandAVCandLRtoVS(VG, sLeftRight, AVC, cLeftRight)

def VSandLRandVCtoVG(VS, sLeftRight, VC):
    """Takes VS and whether the ship goes left or right (-1 or +1),
and VC, then returns VG. Also accepts arrays as inputs."""
    return abs(sLeftRight*VS + VC)

def VSandLRandAVCandLRtoVG(VS, sLeftRight, AVC, cLeftRight):
    """Takes VS, AVC (absolute value of VC), and whether the ship
and current goes left or right (-1 or +1), then returns VG.
Also accepts arrays as inputs."""
    return (VS + sLeftRight*cLeftRight*AVC)

def VSandTAUandVCCONSTtoVG(VS, sLeftRight, tau, vcConst):
    """Takes VS and whether the ship goes left or right (-1 or +1),
and tau and the constants in the VC function, then
constructs VC and uses that to return VG."""
    VC = TAUtoVC(tau, *vcConst)
    if type(VC) == np.ndarray:
        AVC, cLeftRight = VCtoAVCandLRarr(VC)
    else:
        AVC, cLeftRight = VCtoAVCandLR(VC)
    VG = VSandLRandAVCandLRtoVG(VS, sLeftRight, AVC, cLeftRight)
    return VG

def FuncToError(beta, function, inputs, outCorrect):
```

```python
    """This function takes a function (function) and the guess
for its constants (beta), then calculates the error function
E = sum( (realOutput_i - calculatedOutput_i)^2 ) based on
the input values (inputs) and the real outputs (outCorrect)"""
    Sum = 0
    for i in range(len(inputs)):
        Sum += (residual(beta,function,inputs[i],outCorrect[i]))**2
    return Sum


def Error(calculated, real):
    """Takes calculated values and real values, then returns
the error the same way that FuncToError does, except
instead of the function and input, this function
requires the the already calculated values.
ONLY TAKES ARRAYS."""
    return np.sqrt(((calculated-real)**2).sum()/float(len(real)))


def VGtoVSbyMoM(VG):
    """Takes an array of VG, then averages two adjacent
values and returns an array where both those
values are their average."""
    VS = [None]*len(VG)
    for i in range(len(VG)/2):
        vs = (VG[2*i] + VG[2*i + 1])/2.
        VS[2*i] = vs
        VS[2*i + 1] = vs
    return np.array(VS)


def Iteration(abq, vcConst, realParr, VGarr, sLeftRightarr, tauarr,
              method='lm', MoM=False, canBound=True):
    """Does one iteration according to the scheme in the report.
Assumes that the error function has just been dissatisfied,
loops all the way to calculating the new P array. Only takes
arrays as inputs. Returns the new estimate for abq and vcConst,
and the arrays of VS and P according to these constants."""
    # Assume error condition has just been denied. Follow scheme.

    # Set VS' based on MoM, or real P and current a, b, q
    if MoM:
        VSarr = VGtoVSbyMoM(VGarr)
    else:
        VSarr = PtoVS(realParr, *abq)

    if type(VSarr)!=np.ndarray:
        print realParr, abq
```

A

```python
    for vs in VSarr:
        if vs<0:
            print "Here we have some negative VS:"
            print VSarr
    # Set VC' based on VG and VS'
    VCarr = VGandVSandLRtoVC(VGarr, VSarr, sLeftRightarr)
    # Fit VC function
    vcConstNew = ls(residual, vcConst, method=method,
                    args=(TAUtoVC, tauarr, VCarr)).x
    # Set VC based on function
    VCarr = TAUtoVC(tauarr, *vcConstNew)
    # Set VS based on VG and VC
    VSarr = VGandLRandVCtoVS(VGarr, sLeftRightarr, VCarr)
    # Fit P function
    if method=='lm':
        abqNew = ls(residual, abq, method=method,
                    args=(VStoP, VSarr, realParr)).x
    elif method=='trf':
        if canBound:
            abqNew = ls(residual, abq, method='trf',
                        bounds=((-np.inf),
                                (min(realParr), np.inf, 15)),
                        args=(VStoP, VSarr, realParr)).x
        else:
            abqNew = ls(residual, abq, method='trf',
                        args=(VStoP, VSarr, realParr)).x

    # Safeguard for 'a' too big
    aBig = False
    if abqNew[0]>min(realParr):
        aBig = True
    if aBig and canBound:
        abqNew = ls(residual, abq, method='trf',
                    bounds=((-np.inf),
                            (min(realParr), np.inf, 15)),
                    args=(VStoP, VSarr, realParr)).x

    # Set calculated P according to a, b, q, and VS
    calcP = VStoP(VSarr, *abqNew)

    return abqNew, vcConstNew, calcP, VSarr

def IterativeMethod(abq0, vcConst0, realParr, VGarr, tauarr,
                    epsilon=small, timeLimit=timeLimit, method='lm',
                    MoM=False, canBound=True):
```

A

```python
    """"Does the Iterative Method and returns the found constants,
as well as the self-reported error for P, and the number
of loops the process took."""
    # Initialise
    sLeftRightarr = np.array([(-1)**i for i in range(len(VGarr))])
    calcP0 = VStoP(VGarr, *abq0)
    # Set error 1
    EP1 = Error(calcP0, realParr)
    # Do first iteration
    NewRes = Iteration(abq0, vcConst0, realParr, VGarr,
                       sLeftRightarr, tauarr,
                       method=method, MoM=MoM,
                       canBound=canBound)
    abq1, vcConst1, calcP1, VSarr1 = NewRes
    # Set error 2
    EP2 = Error(calcP1, realParr)

    # Loop until error converges
    # Set time limit
    loopNo = 1
    startTime = time.time()
    while abs(EP1-EP2) > epsilon:
##        print "Starting a loop!"
        # Stop prematurely
        loopNo += 1
        elapsedTime = time.time()-startTime
        if loopNo > 10000:
            print "\nToo many loops. Stopped after",
            print elapsedTime, "seconds.\n"
            break
        if elapsedTime > timeLimit:
            print "\nToo much time has elapsed. The time limit is",
            print "set at", timeLimit, "seconds."
            break

        # Do the loop
        EP1 = EP2
        NewRes = Iteration(abq1, vcConst1, realParr, VGarr,
                           sLeftRightarr, tauarr, method=method,
                           canBound=canBound)
        abq1, vcConst1, calcP1, VSarr1 = NewRes
        EP2 = Error(calcP1, realParr)

    return abq1, vcConst1, EP2, loopNo
```

A

```python
def Direct_VGandTAUtoP(abq_vcst0, VG, tau):
    """Takes the constants a,b,q,VCc,VCs,VCt,VC0, and VG and tau, the
directly calculates P from the combination of the formulas
P = a + b*VS^q and VS = VG +/- VC
Accepts arrays as inputs for VG and tau. Also accepts single
values for these as input, but note that VG must then
already contain its sign (-1 if going left, +1 for right)."""
    a = abq_vcst0[0]
    b = abq_vcst0[1]
    q = abq_vcst0[2]
    vccst0 = abq_vcst0[3:]

    if type(VG)==np.ndarray:
        P = np.zeros(len(VG))
        for i in range(len(VG)):
            VCi = TAUtoVC(tau[i], *vccst0)
            VGi = ( (-1)**i )*VG[i]
            try:
                P[i] = a + b*( (abs(VGi - VCi))**q )
            except:
                print "\n"+"-"*20+"Something Happened"+"-"*20+"\n"
                print "a:", a
                print "b:", b
                print "q:", q
                print "VGi:", VGi
                print "VCi:", VCi
                P[i] = a + b*( (abs(VGi - VCi))**q )
    else:
        VC = TAUtoVC(tau, *vccst0)
        P = a + b*( (abs(VG - VC))**q )

    return P

def residual_direct(beta, VGinput, TAUinput, realP):
    """Takes an estimate for the constants (beta), and the input
value(s) for VG and tau, and the real value(s) for P. Then
returns the difference between the calculated P and the
real P. Accepts both arrays for in- and outputs, as well as
single values for them."""
    Poutput = Direct_VGandTAUtoP(beta, VGinput, TAUinput)
    return realP - Poutput

def DirectMethod(abq0, vcConst0, realParr, VGarr, TAUarr,
                 method='lm'):
    """Does the Direct Method."""
```

A

```python
    abq_vccst0_0 = list(abq0) + list(vcConst0)
    try:
        abq_vccst0_1 = ls(residual_direct, abq_vccst0_0,
                          args=(VGarr, TAUarr, realParr),
                          method=method).x
    except RuntimeWarning:
        print "\n------------More info----------------\n"
        print "P:", realParr
        print "VG:", VGarr
        print "TAU:", TAUarr
        abq_vccst0_0[2] = abq_vccst0_0[2]/2
        abq_vccst0_1 = ls(residual_direct, abq_vccst0_0,
                          args=(VGarr, TAUarr, realParr),
                          method=method).x

    Pcalc = Direct_VGandTAUtoP(abq_vccst0_1, VGarr, TAUarr)
    EP = Error(Pcalc, realParr)
    abq1 = abq_vccst0_1[:3]
    vcConst1 = abq_vccst0_1[3:]

    return abq1, vcConst1, EP, 1

def MakeNoise(VG, NoiseOnVG, P, NoiseOnP, tau, NoiseOnTAU):
    """Takes three arrays (VG, P, tau) and puts Measurement Noise
on them, depending on whether they require it (NoiseOn*).
The noise is taken from a normal distribution with mean 0,
and standard deviation a third of the maximum error. These
maximum errors are:
0.05 knots for VG;
25 kWh for P;
36 seconds for TAU."""
    if NoiseOnVG:
        VG_MN = VG + np.random.normal(scale=0.05/3, size=len(VG))
    else:
        VG_MN = VG
    if NoiseOnP:
        P_MN = P + np.random.normal(scale=25./3, size=len(P))
    else:
        P_MN = P
    if NoiseOnTAU:
        tau_MN = tau + np.random.normal(scale=TtoTAU(36)/3,
                                        size=len(tau))
    else:
        tau_MN = tau
```

```python
    return VG_MN, P_MN, tau_MN
```

A.3. PracData

```python
import numpy as np

pracTau = [0, 0.07, 0.12, 0.18, 0.26, 0.33, 0.38, 0.47, 0.52, 0.62]
pracTau = np.array(pracTau)

P11 = [5225, 5225, 6728, 6728, 8716,
       8716, 11503, 11503, 11503, 11503]
V11 = [  14,     14,     15,     15,     16,
         16,     17,     17,     17,     17]
P12 = [2313, 3035, 4018,  5225,  6728,
       8716, 9987, 9987, 11503, 11503]
V12 = [  11,    12,    13,     14,     15,
         16, 16.5, 16.5,     17,     17]
P21 = [34713, 34713, 40478, 40478, 47305,
       47305, 55599, 55599, 65932, 65932]
V21 = [   24,     24,     25,     25,     26,
          26,     27,     27,     28,     28]
P22 = [32141, 34713, 37483, 40478,  43737,
       47305, 51238, 55599, 60466,  65932]
V22 = [ 23.5,     24,  24.5,     25,   25.5,
          26,   26.5,     27,  27.5,     28]
P31 = [1699, 1699, 2094, 2094, 2554,
       2554, 3525, 3525, 5262, 5262]
V31 = [  12,    12,    13,    13,    14,
         14,    15,    15,    16,    16]
P32 = [1483, 1699, 1896, 2094, 2296,
       2554, 2939, 3525, 4304, 5262]
V32 = [11.5,    12,  12.5,    13,  13.5,
         14,  14.5,    15,  15.5,    16]


P11 = np.array(P11)
V11 = np.array(V11)
P12 = np.array(P12)
V12 = np.array(V12)
P21 = np.array(P21)
V21 = np.array(V21)
P22 = np.array(P22)
V22 = np.array(V22)
P31 = np.array(P31)
V31 = np.array(V31)
P32 = np.array(P32)
```

```
V32 = np.array(V32)

pracP = [P11, P12, P21, P22, P31, P32]
pracV = [V11, V12, V21, V22, V31, V32]
```

A.4. TheoData

```python
import numpy as np
from Formulas import *

### Set theoretical values

theoABQ = [[1050.5,  0.8, 2.9],
           [1050.5,  0.8, 2.9],
           [   800, 0.08, 4.2],
           [   800, 0.06, 4.6],
           [  2500, 0.15, 4.7],
           [1050.5,  0.8, 2.9],
           [  2500,  0.8, 2.9],
           [  2500,  0.8, 2.9],
           [  2500,  0.8, 2.9],
           [  2500,  0.8, 2.9],
           [  2500,  0.8, 2.9],
           [  2500,  0.8, 2.9],
           [  2500,  0.8, 2.9]]

theoVS = [[  11,    11,    15,    15,    19,    19,    23,    23,    27,    27],
          [  10,  12.5,    15,  17.5,    20,    20,    20,  22.5,    25,  27.5],
          [  17,    17,    18,    18,    18,    18,    19,    19,    20,    20],
          [  17,    17,    18,    18,    18,    18,    19,    19,    20,    20],
          [  17,    17,    18,    18,    18,    18,    19,    19,    20,    20],
          [  15,    16,    17,    18],
          [  15,  15.5,    16,  16.5,    17,  17.5],
          [  15,  15.5,    16,  16.5,    17,  17.5,    18,  18.5,    19,  19.5],
          [  15,    15,    15,  15.5,  15.5,  15.5,    16,    16,    16,  16.5],
          [  15,    15,    16,    16,    17,    17,    18,    18],
          [  15, 15.25,  15.5, 15.75,    16, 16.25,  16.5, 16.75,    17, 17.25],
          [  16,    18,    20,    22],
          [  16,    18,    20,    22,    16,    18,    20,    22]]

theoTau = [0, 0.07, 0.12, 0.18, 0.26, 0.33, 0.38, 0.47, 0.52, 0.62]

### Make other data from that
sLeftRight = np.array([1,-1]*5)

theoTau = [np.array(theoTau[:len(theoVS[i])])]
```

A

```python
            for i in range(len(theoVS))]
theoTau[12] = np.array([theoTau[12][i/2]
                        for i in range(len(theoVS[12])))])


theoP = [None]*len(theoVS)
for i in range(len(theoVS)):
    theoVS[i] = np.array(theoVS[i])
    theoP[i] = VStoP(theoVS[i], *theoABQ[i])
```