NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Benjaminas Visockis
Ingar Eik Høivik
Thomas Aleksander Jonsson

Distributed Algorithms via Acoustic Marine Communication Gateways

Bachelor's thesis in Electrical Engineering
Supervisor: Pål Holthe Mathisen
May 2024

Benjaminas Visockis
Ingar Eik Høivik
Thomas Aleksander Jonsson

Distributed Algorithms via Acoustic Marine Communication Gateways

Bachelor's thesis in Electrical Engineering
Supervisor: Pål Holthe Mathisen
May 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Preface

This project has given us valuable knowledge and experience in a field that was previously unfamiliar to us. The work has made us understand a lot more about the challenges of acoustics, distributed algorithms and wireless underwater communication, and we are happy to have been able to contribute in the technology development in this field.

We want to thank our clients Damiano Varagnolo and Behdad Aminian for their support in providing technical insight, and motivation to help us make the project as best as possible. We are especially grateful to Emil Wengle for showing up to the lab several times to help us during our development. We also want to thank our supervisor Pål Holthe Mathisen for helping us with the formal aspects of our thesis.

Summary

This thesis presents the development and implementation of an underwater communication system designed to improve distributed optimization algorithms. The project builds upon prior work focused on optimizing Orthogonal Frequency-Division Multiplexing (OFDM) parameters for enhanced communication performance in underwater environments. The primary objective was to physically realize and test the system using underwater acoustic modems.

The project involved three main tasks: refining a cost function to incorporate factors such as noise and packet loss, setting up the communication framework, and integrating the optimization algorithm with the hardware. The resulting system employs the JANUS standard for initial communication to establish optimal parameters before an eventual switch to a higher bit rate protocol for data transfer.

Testing the communication protocol with the optimization algorithm shows potential for convergence in simple optimization problems, and to a certain degree more complex ones as well. Simulating with the new cost function has shown that changing the way it handles packet losses leads to a better outcome.

The thesis demonstrates the feasibility of a modular, distributed optimization-based approach to underwater communication, pushing the technology forward and laying the groundwork for future advancements in autonomous subsea communication systems.

Sammendrag

Denne rapporten presenterer utviklingen og implementasjonen av et undervannskommunikasjonsystem designet for å forbedre distribuerte optimeringsalgoritmer. Prosjektet bygger på tidligere arbeid som fokuserte på å optimere OFDM parametere for å forbedre kommunikasjonsytelse i undervannsmiljøer. Hovedmålet var å fysisk realisere og teste systemet med akustiske modem.

Prosjektet hadde tre hovedmål: forbedre en kostfunksjon til å ta hensyn til faktorer som støy og pakketap, oppsett av et kommunikasjonsrammeverk, og integrering av optimeringsalgoritmen med maskinvaren. Det resulterende systemet bruker JANUS-standaren for initial kommunikasjon som finner optimale parametre før et eventuelt bytte til en protokoll med høyere bit rate for dataoverføring.

Testing av kommunikasjonsprotokollen med optimeringsalgoritmen viser potensiale for konvergens med enkle optimeringsproblemer, og til en viss grad for mer komplekse også. Simulering med den nye kostnadsfunksjonen har vist at å endre måten den håndterer pakketap på leder til et bedre utfall.

Rapporten demonstrerer gjennomførbarheten til en modulær, distribuert optimerings-basert tilnærming for undervannskommunikasjon, som driver teknologien fremover og legger grunnlaget for fremtidige fremskritt innenfor autonome undersjøiske kommunikasjonssystemer.

Table of Contents

List of Figures

List of Tables

List of Acronyms

FH-BFSK	Frequency Hopped Binary Frequency Shift Keying
IPM	Interior Point Method
ISI	Intersymbol Interference
MPSK	M-ary Phase Shift Keying
MQAM	M-ary Quadrature Amplitude Modulation
NATO	North Atlantic Treaty Organization
NL	Noise Level
NRC	Newton-Raphson Consensus
NSL	Noise Spectrum Level
OFDM	Orthogonal Frequency Division Multiplexing
QPSK	Quadrature Phase Shift Keying
ra-NRC	Robust Asynchronous Newton-Raphson Consensus
SDM	Software Defined Modem
sdmsh	Software Defined Modem Shell
SINR	Signal to Interference Plus Noise Ratio
SNR	Signal to Noise Ratio
SWIG	Simplified Wrapper and Interface Generator
TCP	Transmission Control Protocol
UN	United Nations
UWA	Underwater Acoustic(s)
UWAC	Underwater Acoustic Communications

Table 1: Acronyms

1 Introduction

1.1 Background and motivation

Wireless underwater communication is a critical technology for a range of applications including environmental monitoring and underwater exploration. However there are many unique challenges opposed to communication through air. 2.4 GHz electromagnetic waves are commonly used for Wi-Fi on land. If one were to use this underwater then the signal will be completely absorbed by the water at ranges as close as a few centimeters (Schirripa Spagnolo et al. 2020).

Instead one could use low frequency acoustic waves for wireless underwater communication. Water conducts sound waves much better than air, making it possible to achieve transmissions at distances of several thousand meters (EvoLogics 2018). This however also comes with its own problems posed by the underwater environment, such as multi-path propagation, Doppler shifts, and high levels of noise, that make reliable communication a complex problem. Orthogonal Frequency Division Multiplexing (OFDM) is a popular technique used because of its robustness against multi-path effects and its ability to support high data rates (Pérez-Neira and Campalans 2009). Multi-path is when a signal takes multiple paths to the receiver because of the geometry of the environment.

The wave generated by OFDM has different properties based on the parameters given, such as number of subcarriers, symbols per packet and modulation order. It is simple to just pick some parameters to use, but deciding the *best* is trickier because it needs to provide the best bit rate without compromising reliability, and these parameters can vary depending on environmental factors such as noise, depth and distance between transmitter and receiver. By using an optimization algorithm on a cost function describing the wanted effect of the OFDM wave, e.g. bit rate, as a function of the parameters, one is able to find the best parameters. Because of the slow transmission speed, lossy and sometimes unreliable communication, an underwater network is often a distributed system. This means that an optimization algorithm needs to work for a global problem, where each node in the network only has information about its own cost function and the last received data.

JANUS is a standard for underwater acoustic communication developed by NATO. It is considered a robust scheme, suitable for communication in a difficult environment. It suffers however from low bit rate, which makes it not ideal for the transfer of large data streams. There have been suggestions to instead use JANUS in a sort of preliminary stage where it is used to find optimal parameters for a faster protocol, which it later switches to. (Wengle et al. 2024)

1.2 The goal of the project

The goal of the project is to implement an underwater communication scheme that is capable of utilizing distributed optimization algorithms. This project will be based on previous work that had as a goal to find optimal OFDM parameters using distributed optimization. The difference however is that this will be implemented and tested physically using the "S2C R 18/34 USBL Underwater Acoustic Modems". The project also has as a goal to expand on the cost function that is used to calculate the OFDM parameters, such that it can account for factors such as noise and packet loss in a better way. Developing an OFDM implementation and integrating it with the optimization algorithm to test our optimized parameters is the final step.

1.3 Previous work

This work is built upon the work of previous projects that. The code that handles the distributed consensus algorithm was originally written and used in (Iadarola 2022), and it has been given to us by our client for use in this project. The previous groups have shown that JANUS is a viable protocol for transmission, and their work granted us a useful introduction into the use of the protocol.

1.4 Structure of the thesis

This thesis is structured into *Theory*, *Methodology*, *Results*, *Discussion*, and *Conclusion*.

In the theory section concepts such as optimization, distributed optimization, acoustics, OFDM,*sdmsh* and JANUS are explained. The theory is written in such a way that only the parts that are relevant for the project will be brought up, though there may be more technical explanations that, while not utilized directly, can help in understanding how the system works on a deeper level.

Methodology is split into sections where each concerns a ”step” that contributed to building the final product. Because the project was split into three main objectives, the methodology follows a similar fashion of first explaining how the cost function was further developed, then how the communication was set up, and then how they were put together with the optimization algorithm. The last part of methodology explains how the physical tests with the modems were done, both during the development process, and with the finished product

The results section shows experiment data in the form of graphs and numbers with an explanation of what the data means. It also gives an insight into how much the technology in this particular sector has been developed from all the work that has gone into it during this project.

Discussion analyzes the results and brings up different problems that occurred during the development process and also current problems with the finished product. Here there are also theories as to what potentially causes the problems if it is not already known, and proposals of potential solutions that can help others in the future who may want to use this work for their own.

Conclusion brings up in short the relevant points of the thesis and concludes with how the project turned out in the end.

1.5 UN’s goals for sustainable development

There are 17 goals the United Nations have laid that are meant to contribute to a sustainable future. Of these seven goals, there are two that are most relevant for this project.

1.5.1 Industry, Innovation and infrastructure

The United Nations Sustainable Development Goal 9 focuses on building resilient infrastructure, promoting inclusive and sustainable industrialization, and fostering innovation. This drives economic growth, improves efficiency, and addresses environmental challenges. (UN 2023a)

1.5.2 Life below water

United Nations Sustainable Development Goal 14 aims to conserve and sustainably use the oceans, and marine resources. Better knowledge and technology can support sustainable management and conservation of marine resources. (UN 2023b)

2 Theory

2.1 Optimization

In an optimization problem the goal is to either minimize or maximize a cost subject to some constraints. The cost is formulated as a function and is usually called *cost function* or *objective function* (this thesis will use the term *cost function*). The constraints are also functions and are usually called *constraint functions*. This can be formally written as:

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq b_i, \quad i = 1, \ldots, m. \end{array}$$

Where f_0 is the cost function, f_i is the constraint functions for m number of constraints, and x is a vector containing the optimization variables.

There are many types of optimization problems, but this thesis only concerns itself with the type known as *convex optimization*, where only convex functions are used. Graphically speaking, a function is convex if it stays under a line between any two points on the graph of the function (Boyd and Vandenberghe 2004). An example of this is in figure 1.

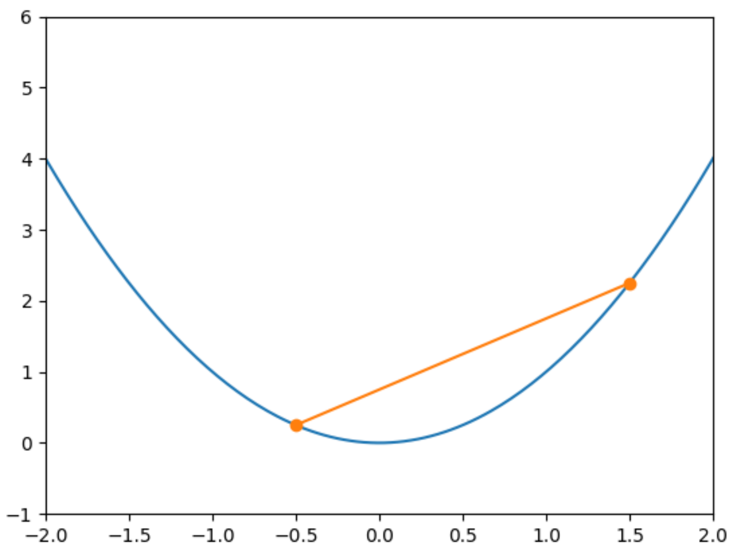


Figure 1: example of a convex function

The logarithmic barrier method is a technique used in optimization for solving constrained optimization problems. It is particularly effective for problems where the constraints are inequalities. The method transforms a constrained optimization problem into an unconstrained problem by introducing a series of barrier functions, one for each constraint. The barrier function adds a growing penalty when the a variable approaches its constraint. A generic cost function using the method is written as:

$$f_0(x) + \sum_{i}^{m} -\frac{1}{t} \log(-f_i(x)) \tag{1}$$

Because the function should be affected as little as possible by the barrier function until close to the constraint, the variable t is introduced. This function becomes more accurate with higher t, as shown in figure 2.

This method is often used with Newton's method (also called the Newton-Raphson method) The downside to choosing a big t is numerical instabilities when a variable approaches a constraint. To circumvent this problem, another method called the "Interior point method" (IPM for short) is often used. In short, this method begins with a low t which it optimizes the problem with. When the cycle is done, t is increased and the algorithm does another round of optimization, with the optimizable variables starting at the solution of the previous IPM-cycle.

everything in this section came from (Boyd and Vandenberghe 2004)

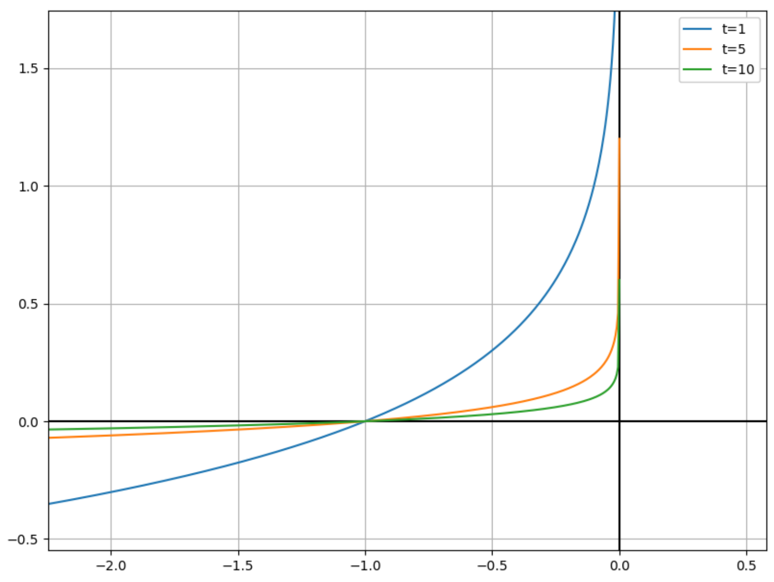


Figure 2: log barrier for different values

2.2 robust asynchronous Newton-Raphson Consensus

In (Varagnolo et al. 2016) a method was developed to achieve distributed convex optimization using Newton-Rhapson. In (Bof et al. 2019) (which is the source for the entire section) it was expanded to cope with asynchronous communication and lossy transmissions by combining three different building blocks: *Newton-Raphson consensus*, *push-sum algorithm* and *robust ratio consensus*. While understanding every part of the theory in this section is not necessary for the thesis, it does explain a bit of the thought behind the algorithm that was used. The full pseudocode for the combined algorithm can be found in the same paper, but parts of it will be shown later in Methodology when they are relevant.

2.2.1 Notation

A glossary for some of the notation:

Symbol	Meaning
$\mathcal{C}^3$	function that has a third order continuous derivative
∇^2	Hessian matrix
ε	step size
x^+	simplified way to write $x(k+1)$ for $x(k)$
$\rho_{i,y}^{(j)}$	a variable that contains y-data that node i received from node j
$\mathcal{N}_i^{\text{in}}$	the set of neighboring nodes that node i receives from
$\mathcal{N}_i^{\text{out}}$	the set of neighboring nodes that node i sends to
$\lvert\mathcal{N}_i^{\text{in/out}}\rvert$	the number of elements in the set (for in- and out-neighbors respectively)
$\mathcal{V}$	the set of all the nodes in the communication network

2.2.2 Assumptions

The problem is formulated as:

$$x^* := \underset{x}{\operatorname{argmin}}\ f(x) = \underset{x}{\operatorname{argmin}} \sum_{i=1}^{N} f_i(x) \tag{2}$$

Where $x \in \mathbb{R}^n$ and where the local costs $f_i\ :\ \mathbb{R}^n \mapsto \mathbb{R}$ satisfies the following assumptions:

Assumption 1 (Cost smoothness): Each f_i is known only to node i, is $\mathcal{C}^3$, and is strongly convex, meaning its Hessian is bounded from below: $\nabla^2 f_i(x) > cI_n$ for all x, with $c > 0$ being some positive scalar.

Assumption 2 (Network connectivity): The communication graph among the nodes is fixed, directed, and strongly connected, i.e., for each pair of nodes, there is at least one directed path connecting them.

2.2.3 Newton-Raphson consensus

Newton-Raphson consensus is based on the fact that in a centralized scenario, the standard Newton-Rhapson update can be written as:

$$\begin{aligned} x^+ &= x - \varepsilon \left(\nabla^2 f(x)\right)^{-1} \nabla f(x) \\ &= (1-\varepsilon)x + \varepsilon \left(\textstyle\sum_i \nabla^2 f_i(x)\right)^{-1} \left(\textstyle\sum_i \left(\nabla^2 f_i(x)x - \nabla f_i(x)\right)\right) \end{aligned} \tag{3}$$

Where:

$$\begin{aligned} h(x) &:= \textstyle\sum_i \nabla^2 f_i(x) \\ g(x) &:= \textstyle\sum_i \left(\nabla^2 f_i(x)x - \nabla f_i(x)\right) \end{aligned} \tag{4}$$

If all nodes can have a different x_i then:

$$x_i^+ = (1-\varepsilon)x_i + \varepsilon \left(\sum_j \nabla^2 f_j(x_j)\right)^{-1} \left(\sum_j \left(\nabla^2 f_j(_j)x_j - \nabla f_j(x_j)\right)\right) \tag{5}$$

Where:

$$\begin{aligned} h_j(x_j) &:= \nabla^2 f_j(x_j) && \text{and} \quad \overline{h}(x_1,\ldots,x_N) := \textstyle\sum_j h_j(x_j) \\ g_j(x_j) &:= \nabla^2 f_j(x_j)x_j - \nabla f_j(x_j) && \text{and} \quad \overline{g}(x_1,\ldots,x_N) := \textstyle\sum_j g_j(x_j) \end{aligned} \tag{6}$$

The dynamics of the N local systems are identical and exponentially stable. Because they are all driven by the same forcing term $\kappa(x_1,\ldots,x_n) = \left(\overline{h}(x_1,\ldots,x_N)\right)^{-1} \overline{g}(x_1,\ldots,x_N)$, it is expected that:

$$x_i - x_j \to 0 \quad \forall i,j$$

Which implies that all the local variables will be identical.

2.2.4 Push-Sum Consensus

The Newton-Raphson consensus requires that each node can compute the two sums $z_i = \overline{h}$ and $y_i = \overline{g}$ at least asymptotically. A ratio of the two are needed, and each node can asymptotically converge to a scaled version of the two; in other words:

$$y_i \to \eta_i \overline{g}(x_1, \dots, x_N) = \eta_i \sum_j g_j(x_j) \tag{7}$$

$$z_i \to \eta_i \overline{h}(x_1, \dots, x_N) = \eta_i \sum_j h_j(x_j) \tag{8}$$

Where $\eta_i, \dots, \eta_N$ are possibly time-dependent nonzero scalars. The arrow means that the difference between left and right side becomes 0 as the iterations approaches infinity. This can be expanded to an asynchronous scenario. If at any time k only one node i is activated and it updates its variables and broadcasts them to its neighbors, and then, consistently, the receiving node j updates its local variables, the update rules for y_i and y_j become:

$$\begin{aligned} y_i^+ &= \tfrac{1}{|\mathcal{N}_i^{\text{out}}|+1} y_i \\ y_j^+ &= \tfrac{1}{|\mathcal{N}_j^{\text{out}}|+1} y_i = y_j + y_i^+ \quad \forall j \in \mathcal{N}_i^{\text{out}} \end{aligned} \tag{9}$$

If the activation of the nodes is randomized, independent and identically distributed, the local variables converge to:

$$y_i \to \eta_i(k) \sum_i y_i(0) = \eta_i(k) \sum_i g_i(x_i) = \eta_i(k) \overline{g}(x_1, \dots, x_N) \tag{10}$$

$$z_i \to \eta_i(k) \sum_i z_i(0) = \eta_i(k) \sum_i h_i(x_i) = \eta_i(k) \overline{h}(x_1, \dots, x_N) \tag{11}$$

Where $\eta_i(k) > 0$ depends on the activation sequence of the nodes.

2.2.5 Robust Ratio Consensus

The previous algorithm does not converge in the case of packet loss. A solution to this issue is to add additional variables $\sigma_{i,y}, \rho_{i,y}^{(j)}$ that acts as mass counters. $\sigma_{i,y}$ acts as the mass counter for the total y-mass sent to its neighbors. $\rho_{i,y}^{(j)}$ acts as the mass counter for the total y-mass received for each in-neighbor $j \in \mathcal{N}_i^{in}$. The update becomes:

$$\sigma_{i,y}^+ = \sigma_{i,y} + y_i \quad \forall i \in \mathcal{V} \tag{12}$$

$$\rho_{i,y}^{(j)+} = \begin{cases} \sigma_{j,y} & \text{if } \sigma_{j,y} \text{ is received} \\ \rho_{i,y}^{(j)} & \text{otherwise} \end{cases} \quad \forall_j \in \mathcal{N}_i^{in} \tag{13}$$

$$y_i^+ = \frac{1}{|\mathcal{N}_i^{\text{out}}|+1} y_i + \sum_{j \in \mathcal{N}_i^{\text{in}}} \frac{1}{|\mathcal{N}_j^{\text{out}}|+1} \left(\rho_{i,y}^{(j)+} - \rho_{i,y}^{(j)} \right) \tag{14}$$

If during iteration k, node i receives information from node j, the information related to node j used to update of the variable y_i is $\nu_{i,y}^{(j)}(k) = \sigma_{j,y} - \rho_{i,y}^{(j)}$. The variable $\nu_{i,y}^{(j)}(k)$ is a virtual mass that, under reliable transmission, is zero. If packet loss occurs, the variable accumulates mass that node j attempts to transfer to node i. In this way the mass is not lost, making the following valid for each time instant k:

$$\sum_i \left(y_i(k) + \sum_{j \in \mathcal{N}_i^{\text{in}}} \nu_{i,y}^{(j)}(k) \right) = \sum_i y_i(0) \tag{15}$$

2.3 Acoustics

Acoustics is the field of science concerned with sound. This section will go over some important concepts in this field.

2.3.1 Sound pressure and intensity

Sound pressure, denoted p and measured in pascal (Pa), is the amount of force over an area:

$$p = \frac{F}{A} \tag{16}$$

Sound intensity, measured in watts per square meter (W/m^2), is the power transmitted per time unit through a unit area in a specified direction:

$$I = \frac{p^2}{\rho c} \tag{17}$$

where ρc (density times speed of sound) is the acoustic impedance of the transport medium. (Coates 1990; Dosits n.d.(a))

2.3.2 Decibels and reference values

Because of the wide range of relevant sound pressure levels that exist, sound levels are often measured in terms of decibels, meaning a logarithmic scale relative to some reference value. The formula to convert a value into its relative decibel representation is:

$$x^{\text{dB}} = 10 \log_{10} \left(\frac{x}{x_0} \right) \tag{18}$$

Which is often written in the form "x^{dB} dB re x_0", where x_0 is the reference value.

For sound underwater under water, the standard reference value p_0 is 1 μPa. This makes the reference intensity $I_0 = 0.67 \cdot 10^{-18}\ W/m^2$

The decibel representation of sound intensity (also called sound level) can also be calculated directly from the measured pressure:

$$I^{\text{dB}} = 10 \log_{10} \left(\frac{p^2}{p_0^2} \right) = 20 \log_{10} \left(\frac{p}{p_0} \right) \tag{19}$$

(Coates 1990); (Dosits n.d.[a])

2.3.3 Source level

Source level is a metric describing the power radiating from a source in a particular direction. It is a key component of the sonar equation and can be combined with transmission loss to calculate received levels. Although similar to sound intensity, it differs in that it is not dependent on the distance away from the source or on the propagation environment surrounding the source. It can be thought of as the relative sound intensity at a 1 meter distance from an infinitely small source in a specific direction, and it is expressed in "dB re 1 μPa". (Dosits n.d.[b])

Regarding a source with spherical spreading the area the power radiates through is $4\pi r^2 \approx 12.6\ m^2$. If the source output is P watts and it radiates equally strongly in all directions, the source level can be calculated with the following formula[1]:

$$\mathrm{SL} = 10\log_{10}\left(\frac{P}{12.6 \cdot 0.67 \cdot 10^{-18}}\right) = 10\log_{10}(P) + 170.8 \tag{20}$$

(Coates 1990)

2.3.4 Noise and interference

Noise is considered unwanted power that is picked up by a receiver in addition to the useful power containing data. Noise is often given in terms of power spectral density N_0, which is the measure of noise power per Hz. This value is multiplied by the transmission bandwidth B to get the noise power. SNR is defined as the ratio between the received signal power P and the power of the noise, and have to be the same unit (for example watts):

$$\mathrm{SNR} = \frac{P}{N_0 B} \tag{21}$$

In cases where there is interference I, SINR is used instead of SNR:

$$\mathrm{SINR} = \frac{P}{N_0 B + I} \tag{22}$$

SNR and SINR are often used to estimate bit errors in a signal.

Intersymbol interference (ISI) is when a symbol interferes with another. One of the causes for this is called multipath, and it is when a symbol reflects of something and goes into the receiver as a delayed version of the sent signal. Because increasing the transmitter power also increases the interference, it causes an irreducible error floor, making the SINR cap at a certain value. (Goldsmith 2005)

2.3.5 Underwater background noise

A model for typical sources of background noise that is encountered in the deep sea, such as turbulence, shipping, surface agitation, and thermal noise, was researched in (Wenz 1962). This also gave rise to what's known as Wenz curves, which is the graphical representation of the models. The mathematical formula for the different sources are as follows:

$$\begin{aligned}
&\mathrm{NSL}_1 = 17 - 30\log_{10}(\mathbf{f}) && \text{thermal}\\
&\mathrm{NSL}_2 = 40 + 20(s + 0.5) + 26\log_{10}(\mathbf{f}) - 60\log_{10}(\mathbf{f} + 0.03) && \text{shipping}\\
&\mathrm{NSL}_3 = 50 + 7.5\mathrm{w}^{1/2} + 20\log_{10}(\mathbf{f}) - 40\log_{10}(\mathbf{f} + 0.4) && \text{surface agitation}\\
&\mathrm{NSL}_4 = -15 + 20\log_{10}(\mathbf{f}) && \text{thermal}
\end{aligned} \tag{23}$$

The models and the curve shown in figure 3 shows the Noise spectral level (NSL) for each source as a function of kHz, where NSL is the decibel representation of the power spectral density of the noise.

[1]In the reference it says 167, but doing the same calculations yields a more accurate 170.8

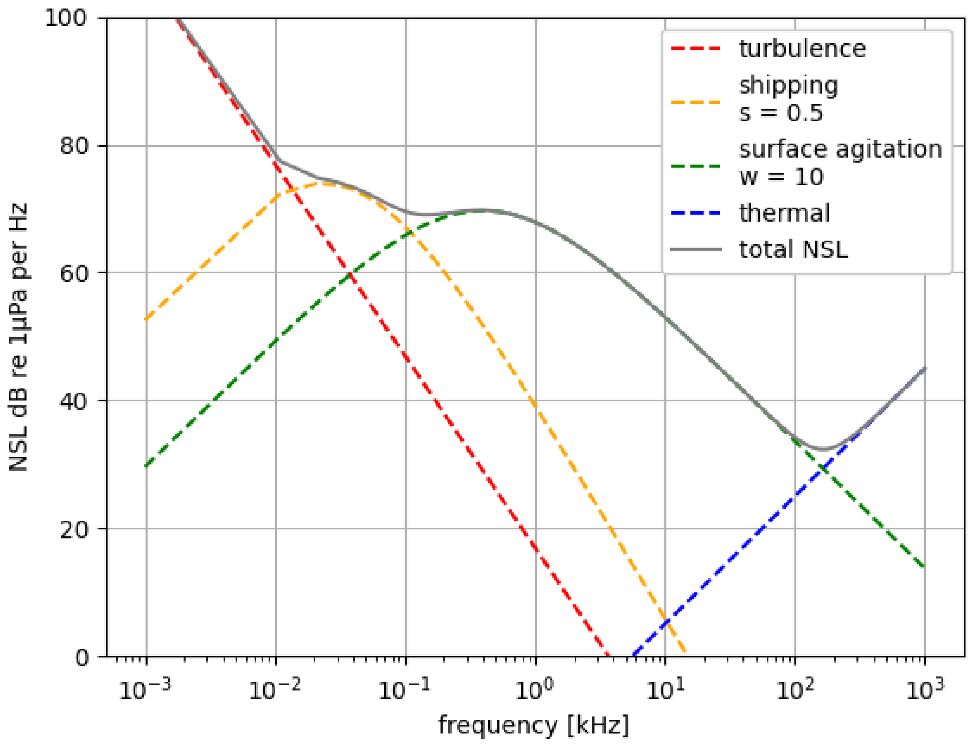


Figure 3: Example of a Wenz curve with moderate shipping (s = 0.5) and light breeze (w = 10)

s is a number between 0 and 1 describing shipping density, w is wind speed in m/s, and $\boldsymbol{f}$ is the frequency in kHz.

The total NSL is calculated by converting each NSL from decibel, adding them, and converting back:

$$\mathrm{NSL} = 10\log_{10}(10^{\mathrm{NSL}_1/10} + 10^{\mathrm{NSL}_2/10} + 10^{\mathrm{NSL}_3/10} + 10^{\mathrm{NSL}_4/10}) \tag{24}$$

NSL is multiplied with the bandwidth to get just the noise level (NL), which in decibel terms means adding the logarithm of the bandwidth:

$$\mathrm{NL} = \mathrm{NSL} + 10\log_{10}(\mathrm{B}) \tag{25}$$

Calculations from (Coates 1990).

2.3.6 Absorption

An acoustic signal will experience a loss depending on the frequency and distance it travels. One model for underwater absorption was developed in (Fisher and Simmons 1977), which is usually called the Fisher & Simmons model. This model gives an absorption coefficient in dB/km based on the frequency in kHz ($\mathbf{f}$), temperature in Celsius (t), and depth in meters (d):

$$10\log_{10}(\alpha(\mathbf{f})) = A_1 P_1 \frac{f_1 \mathbf{f}^2}{f_1^2 + \mathbf{f}^2} + A_2 P_2 \frac{f_2 \mathbf{f}^2}{f_2^2 + \mathbf{f}^2} + A_3 P_3 \mathbf{f}^2 \tag{26}$$

Where:

$$\begin{aligned}
A_1 &= 1.03 \cdot 10^{-8} + 2.36 \cdot 10^{-10} \cdot t - 5.22 \cdot 10^{-12} \cdot t^2 \\
A_2 &= 5.62 \cdot 10^{-8} + 7.52 \cdot 10^{-10} \cdot t \\
A_3 &= (55.9 - 2.37 \cdot t + 4.77 \cdot 10^{-2} \cdot t^2 - 3.48 \cdot 10^{-4} \cdot t^3) \cdot 10^{-15} \\
f_1 &= 1.32 \cdot 10^3 \cdot (t + 273.1) \cdot \exp(\tfrac{-1700}{t+273.1}) \\
f_2 &= 1.55 \cdot 10^7 \cdot (t + 273.1) \cdot \exp(\tfrac{-3052}{t+273.1}) \\
P_1 &= 1 \\
P_2 &= 1 - 10.3 \cdot 10^{-5} \cdot d + 3.7 \cdot 10^{-9} \cdot d^2 \\
P_3 &= 1 - 3.84 \cdot 10^{-5} \cdot d + 7.57 \cdot 10^{-10} \cdot d^2
\end{aligned} \tag{27}$$

Formula and coefficients from (Sehgal et al. 2009)

2.4 Interfacing with the modems

One may wish to send several different commands to a modem, e.g start transmitting, start receiving or report errors. For EvoLogics modems, which are the ones exclusively used in this thesis, these commands must take the form of a frame with the following format:

```
<0x80,0x00,0x7f,0xff,0x00,0x00,0x00,0x00>
      ,<cmd:8>,<param:24>,<len:32>,<data:len*16>
```

This frame is to be sent using ethernet over a modems default socket; 4200. The different parts of the frame enclosed by <> describe the different parameters of a command. For example, the *<cmd:8>* part of the frame specifies the nature of the command, whether it's a transmission command or a reception command for example.

2.4.1 sdmsh

The user doesn't have to interact with frames directly, one can instead use *sdmsh*. *sdmsh* (Software Defined Modem Shell) is a command line tool and library provided by EvoLogics, found on GitHub (https://github.com/EvoLogics/sdmsh). *sdmsh* allows for easy interfacing with an EvoLogics modem, giving the user atomic control over the modem.

The terminal tools provide a great method for basic communication and debugging. These terminal commands also provide the method for which the modems are initially configured upon activation. Six different terminal commands are commonly used when working on the modems. The first one is simply the method of connecting to a modem:

```
./sdmsh IP
```

where *./sdmsh* is the directory of the installation, and *IP* is the modem's IP-address. This generates a session with the modem and you will be put in a interactive shell. Only one session may be active at the time, if a different user tries to connect to the modem it will terminate the prior session. After successfully creating a session one may begin using *sdmsh* to send commands. An important commands is:

```
stop
```

which must be run in order to put the modem in SDM mode (Software Defined Modem mode), or else the modem will ignore further commands. The next important command is

```
config <threshold> <gain> <source level> [<preamp_gain>]
```

which specifies how the modem will do the signal processing. Only two of these values are modified during work; <*threshold* > and <*source level* >. Selecting a low <*threshold* > of 30 will make the modem sensitive to detection of preambles and may trigger false positives. A value of 350 would make the modem robust to noise but require a loud and clear signal to trigger detection. *<source level >* is a discrete setting with four different values, and specifies the power level of the modem, where 0 is max power and 3 is minimum power. Switching between different power levels and one can hear an audible difference in the loudness of a transmission, one will also observe a significant difference in current usage. A power level of 3 will maximally draw around 0.15 amps, while a power level of 0 will draw over 2 amps. How much more than 2 amps it will draw is unclear, as the laboratory power supply used only had a max current of 2 amps.

Before transmitting and receiving, one must run the following command on the receiver:

```
ref [<number of samples>] [<driver>:]<params>
```

This sets a preamble for the modem to listen for. A preamble is how a modem knows it's receiving a transmission meant for itself. *[< driver >:]< params >* specifies the path or origin of the signal one wishes to use as a preamble. *[number of samples]* specifies the length of the preamble, where a single sample is 16 bits. Now if one wishes to transmit to this modem, one must preface the transmitted signal with the same preamble. To start listening for transmissions, run:

```
rx <number of samples> [<driver>:]<params> [[<driver>:]<params>]
```

< number of samples > decides how many samples you wish to listen for after receiving a preamble. *[< driver >:]< params >* gives you some flexibility in how you want the received signal to be stored. The simplest method is to specify a file path and the modem will fill up the designated path with it's received samples. To transmit use:

```
tx [<number of samples>] [<driver>:]<parameter>
```

< number of samples > decides how many samples one wishes to transmit. *[< driver >:]< params >* will be the path of the signal you wish to transmit. This signal has to be concatinated with the same preamble the receiver uses, or the receiver will ignore the transmission.

2.4.2 sdmsh and Python

For realtime communication we move away from typing terminal commands, and instead automate data transmission and reception using programming. While the underlying code of the *sdmsh* is written in "C", EvoLogics have included a interface file called *sdm.i*, which allows the user to create a SWIG implementation. SWIG (Simplified Wrapper and Interface Generator) can be used to allow Python to run "C" functions. Which functions to convert is specified by EvoLogics in *sdm.i*. By running

```
swig -python sdm.i
```

the Python libraries will be created. Now one may interface with the modem using Python. The newly genrated Python functions will in most cases follow the same principles of the command line commands. As we have moved away from shell commands, to differ between referencing the Python libraries instead of the command line commands we use *sdm* instead of *sdmsh*. The different functions are explored in detail in Methodology.

2.5 JANUS

ANEP-87, also called JANUS, is a standard for underwater acoustic communication proposed by NATO (NATO 2024). It was developed out of the need for a standard of communication both between military and civilian maritime assets. The standard describes a baseline JANUS packet - an acoustic waveform consisting of 64 bits of information, with the facility to append almost arbitrary cargo data to the end of the packet. The coding scheme uses the robust Frequency Hopped Binary Frequency Shift Keying (FH-BFSK) to ensure signal robustness in underwater conditions and simplicity of implementation. The encoding process for a baseline JANUS packet is shown in Figure 4.

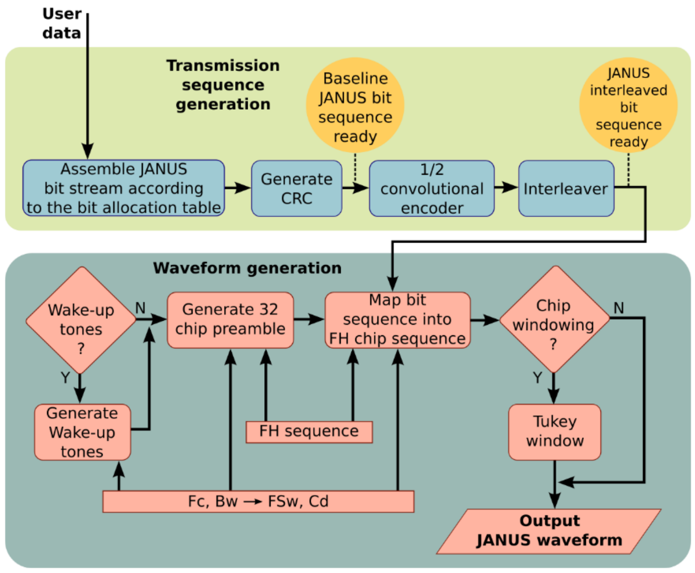

User data
Transmission sequence generation
Baseline JANUS bit sequence ready
JANUS interleaved bit sequence ready
Assemble JANUS bit stream according to the bit allocation table
Generate CRC
1/2 convolutional encoder
Interleaver
Waveform generation
Wake-up tones ?
N
Y
Generate 32 chip preamble
Map bit sequence into FH chip sequence
Chip windowing ?
N
Y
Generate Wake-up tones
FH sequence
Tukey window
Fc, Bw → FSw, Cd
Output JANUS waveform

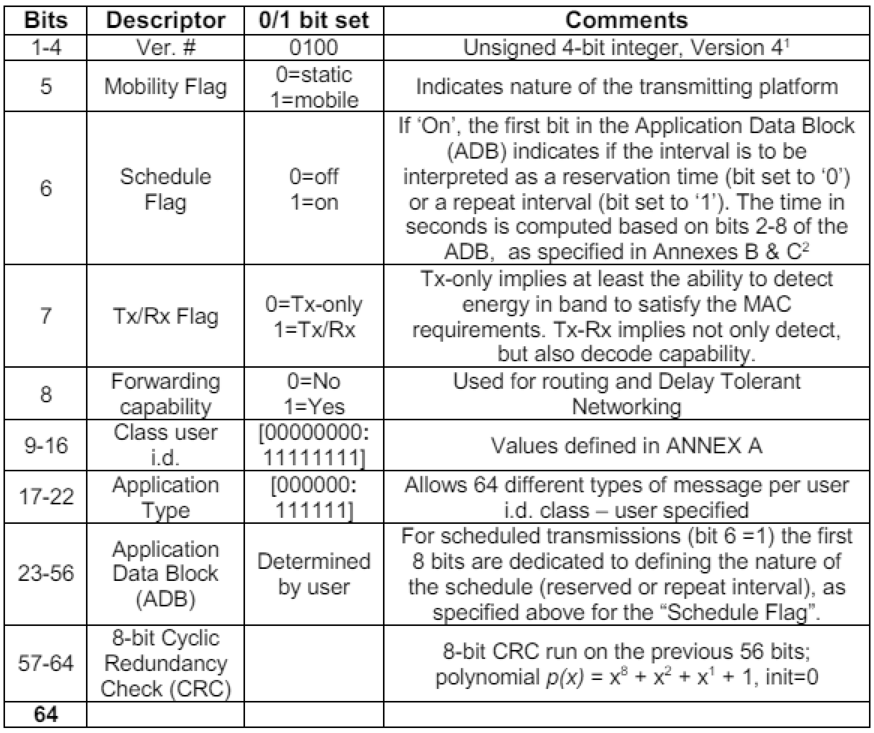
Figure 4: Block-diagram showing the generation of a baseline JANUS packet (NATO 2024)
.

With the inclusion of cargo data in addition to the baseline JANUS packet, unless specified otherwise, the coding scheme continues to encode the cargo data using the same convolutional encoder and interleaver as the baseline JANUS packet. As shown in Figure 4, the encoding process is complicated and consists of a lot of steps. Understanding each part of the encoding process goes beyond the span of this thesis, as the existing implementations of JANUS allows the user to generate packages from a handful of parameters, never having to interact with each individual part of the encoding process.

When using JANUS you are limited to a data rate of 36 to 69 bit/s (depending on cargo length) in the 10 kHz area (Wengle et al. 2024), thus JANUS is not ideal for operations which require high data rates. Through (Wengle et al. 2024) it is suggested to use JANUS as a preliminary protocol - using the benefits of its standardization and robustness. As a preliminary protocol it can be used to decide the parameters of further communication, e.g two nodes make first contact, then communicate using JANUS to decide the channel parameters and which communication protocol to use for further communication.

2.5.1 The JANUS packet

The NATO documentation (NATO 2024) describes the JANUS packet in detail. A JANUS packet starts with a fixed preamble of 32 symbols length. This preamble is used for signal detection and synchronization. Following the fixed preamble is the baseline JANUS packet of 64 bits, with the following bit allocation:



Bits	Descriptor	0/1 bit set	Comments
1-4	Ver. #	0100	Unsigned 4-bit integer, Version 4[1]
5	Mobility Flag	0=static 1=mobile	Indicates nature of the transmitting platform
6	Schedule Flag	0=off 1=on	If 'On', the first bit in the Application Data Block (ADB) indicates if the interval is to be interpreted as a reservation time (bit set to '0') or a repeat interval (bit set to '1'). The time in seconds is computed based on bits 2-8 of the ADB, as specified in Annexes B & C[2]
7	Tx/Rx Flag	0=Tx-only 1=Tx/Rx	Tx-only implies at least the ability to detect energy in band to satisfy the MAC requirements. Tx-Rx implies not only detect, but also decode capability.
8	Forwarding capability	0=No 1=Yes	Used for routing and Delay Tolerant Networking
9-16	Class user i.d.	[00000000: 11111111]	Values defined in ANNEX A
17-22	Application Type	[000000: 111111]	Allows 64 different types of message per user i.d. class – user specified
23-56	Application Data Block (ADB)	Determined by user	For scheduled transmissions (bit 6 =1) the first 8 bits are dedicated to defining the nature of the schedule (reserved or repeat interval), as specified above for the "Schedule Flag".
57-64	8-bit Cyclic Redundancy Check (CRC)		8-bit CRC run on the previous 56 bits; polynomial $p(x) = x^8 + x^2 + x^1 + 1$, init=0
64			

Figure 5: Bit allocation table of a baseline JANUS packet (NATO 2024)

The baseline packet is encoded using a convolutional encoder with a 2:1 redundancy ratio. The convolutional encoder follows the IS-95 CDMA standard. After encoding, the message will have increased to 144 bits, where 64 bits are information. Then the message gets interleaved. The interleaver separates each consecutive bits by a depth value, D, which is a function of the message size. By reordering and spreading out consecutive bits, you reduce the severity of burst errors, and combined with convolution encoding, provides a robust signal. The baseline JANUS packet and the additional cargo are interleaved separately.

2.5.2 JANUS Tool Kit 3.0.5

The NATO Centre for Maritime Research and Experimentation released a sample JANUS implementation to the community, called the JANUS Tool Kit 3.0.5. It is made available through the JANUS Wiki (https://januswiki.com/). It contains two implementations, but the relevant one is the one written in the "C" programming language. The "C" implementation consists of two command line executables *janus-tx* and *janus-rx*, and an underlying "C" library. Through *janus-tx* you gain the ability to create JANUS packet waveforms from a list of parameters set by the user. To list some of them: central frequency, sampling frequency, binary format representations, cargo data and output format e.g *.raw* or *.wav*. Using *janus-rx* you are able to decode a JANUS waveform on the receiving side. Similarly to *janus-tx*, in *janus-rx* there are several parameters to choose from. In order to decode a JANUS packet correctly, parameters like central frequency, sampling frequency and binary format representations should be set equal to the parameters of *janus-tx*.

Though *janus-tx* and *janus-rx* provide simplicity to packet generation and decoding, one may wish to move away from command line implementations and instead interact with the underlying library, as you don't want to pass command line arguments each time you want to send and receive packets. The method used for continuous packet encoding and decoding is described in Methodology.

2.6 Digital Signal Processing

Digital Signal Processing (DSP) is a field of study that consists of analysis, modification, and synthesis of digital signals. It involves analog signals, which are continuous in time and amplitude, and converting them to- and from discrete time domain, in which they can be handled by digital devices.

2.6.1 Signal Modulation

In order to transmit data over the underwater acoustic (UWA) channel, the data to be transmitted needs to be modulated. Modulation is the process of taking some digital data, usually present in the form of binary code (ones and zeros) and assigning it to a sinusoidal carrier signal, by varying one or more its properties, such as frequency (f), amplitude (α), and phase (θ). The resulting signal can be given by

$$s(t) = \alpha(t)\cos[2\pi(f_c + f(t))t + \theta(t) + \phi_0], \tag{28}$$

where f_c is the carrier frequency, ϕ_0 is phase offset of the carrier, and $\alpha(t)$, $f(t)$, $\theta(t)$ are the time-varying modulated data amplitude, frequency, and phase, respectively. A visual representation of the signal waveforms is shown in Figure 6, where (a) is the high-frequency carrier signal, (b) is the data signal, (c), (d), and (e) are the carrier and data signals modulated using amplitude, frequency, and phase shift keying, respectively.

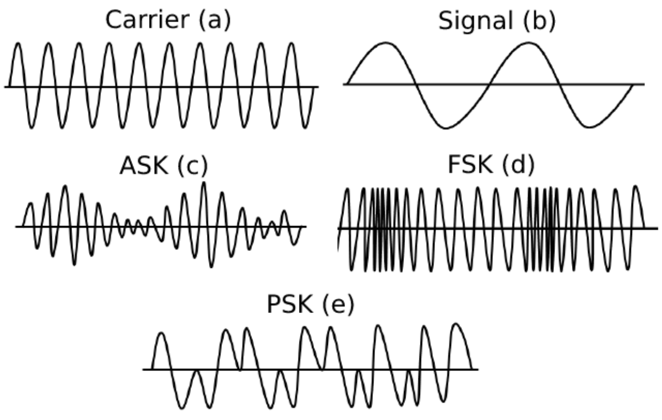


Figure 6: Simple example of modulated signal (Cho and Woo 2011)

However, single degree-of-freedom modulation techniques, such as in Figure 6 are limiting, as they do not fully utilize the information capacity of their respective signals. By using a technique called I/Q modulation it is possible to represent a signal using two components: the in-phase (I) and quadrature (Q) components. These components are orthogonal to each other, with the Q component being 90 degrees out of phase with the I component. By modulating the amplitudes of the I and Q components, the amplitude, frequency, and phase of the resulting modulated signal can be modified (Kuisma 2023). This I/Q signal can be represented visually in Figure 7 and mathematically by rewriting Equation 2.6.1 in terms of its I and Q components:

$$\begin{aligned} s(t) &= \alpha(t)\cos(\phi(t) + \phi_0)\cos(2\pi f_c t) - \alpha(t)\sin(\phi(t) + \phi_0)\sin(2\pi f_c t) \\ &= s_I(t)\cos(2\pi f_c t) - s_Q(t)\sin(2\pi f_c t), \end{aligned}$$

where $\phi(t) = 2\pi f(t)t + \theta(t)$, and s_I and s_Q are the in-phase and quadrature components of $s(t)$, respectively.

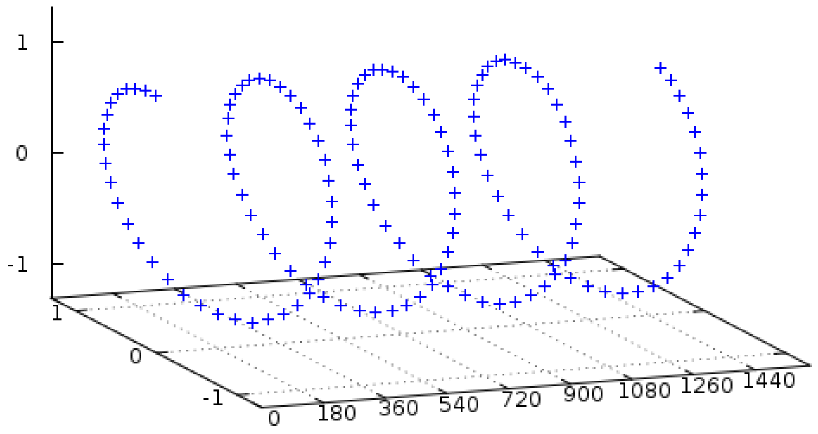


Figure 7: Sinusoidal I/Q signal shown in 3D (Kuisma 2023)

For this project, the M-ary Quadrature Amplitude Modulation (MQAM) is used. MQAM is a modulation technique that combines both amplitude and phase modulation to transmit data efficiently over the communication channel. In MQAM, the binary data is mapped to a constellation of M symbols, where each symbol represents a unique combination of amplitude and phase values. The value of M is a power of 2, usually between 2^2 and 2^6, represented by the following diagram:

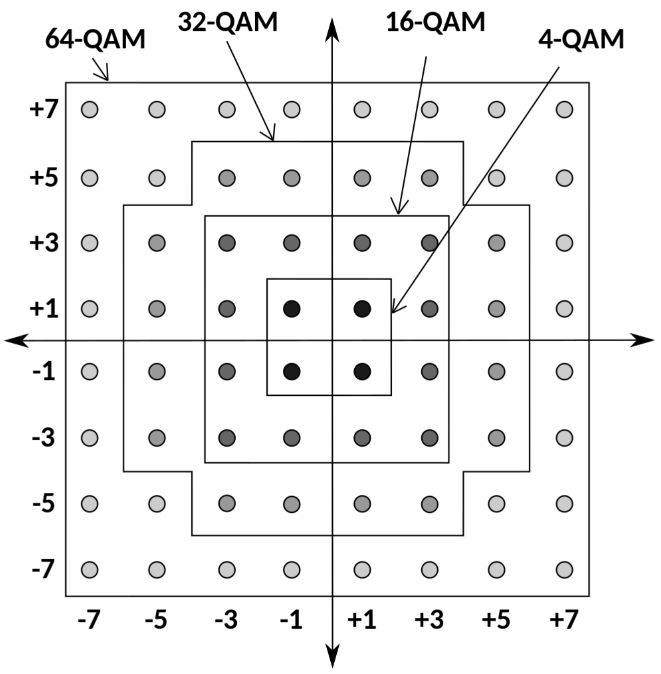


Figure 8: Rectangular constellation for QAM

The two degrees-of-freedom in MQAM allow it to take better advantage of the frequency spectrum and more efficiently encode data than one-dimensional techniques which modulate either amplitude or phase only (Goldsmith 2005, Chapter 5.3.3). By utilizing both the I and Q components of the signal, MQAM can encode more bits per symbol, leading to higher spectral efficiency.

The MQAM modulation process can be described mathematically as follows:

$$s(t) = \alpha(t)\cos(2\pi f_c t + \theta(t)) \tag{29}$$

The amplitude and phase values are determined by the location of the symbol in the MQAM constellation diagram (Figure 8).

The bandwidth utilization with MQAM increases with the modulation order M, as more bits can be transmitted per symbol. The number of bits per symbol, k, is related to the modulation order by:

$$k = \log_2(M) \tag{30}$$

However, higher-order modulations require a higher SNR to maintain an acceptable BER, as the symbols become more closely spaced in the constellation diagram (Chen and Wyglinski 2010).

2.6.2 Orthogonal Frequency Division Multiplexing

Orthogonal Frequency Division Multiplexing (OFDM) is a technique of utilizing the available frequency bandwidth by dividing it into multiple carrier signals, called subcarriers (denoted with N). They are transmitted simultaneously in different frequency bands and are mathematically orthogonal to each other in the frequency domain - the amplitude peak of each subcarrier coincides with the nulls of the adjacent subcarriers (Figure 9). Each of them carry their own portion of the data stream modulated using a technique such as QPSK, MPSK, MQAM, or similar.

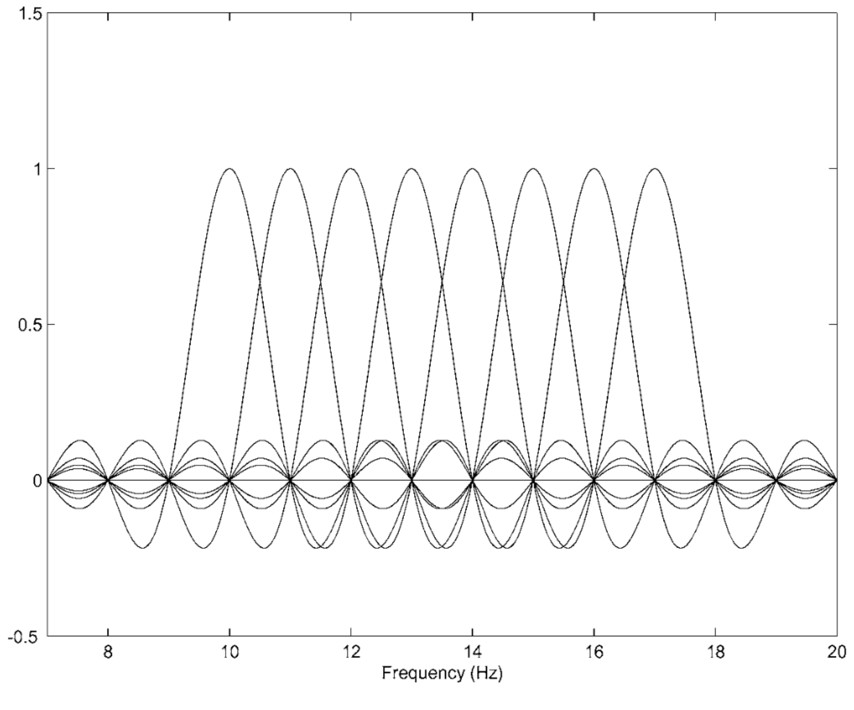


Figure 9: OFDM overlapping subcarriers for rectangular pulses with $f_0 = 10$ Hz and $\Delta f = 1$ Hz (Goldsmith 2005)

The purpose of OFDM is to pack in as much data as possible whilst preserving maximum possible data rate and trying to minimize interference from various sources of noise (Section 2.3.4). This property allows for efficient use of the available spectrum and makes data transmission robust against interference in difficult propagation channels, such as multipath (Firdaus and Moegiharto 2022).

The OFDM (de)modulation process can be described by the following steps:

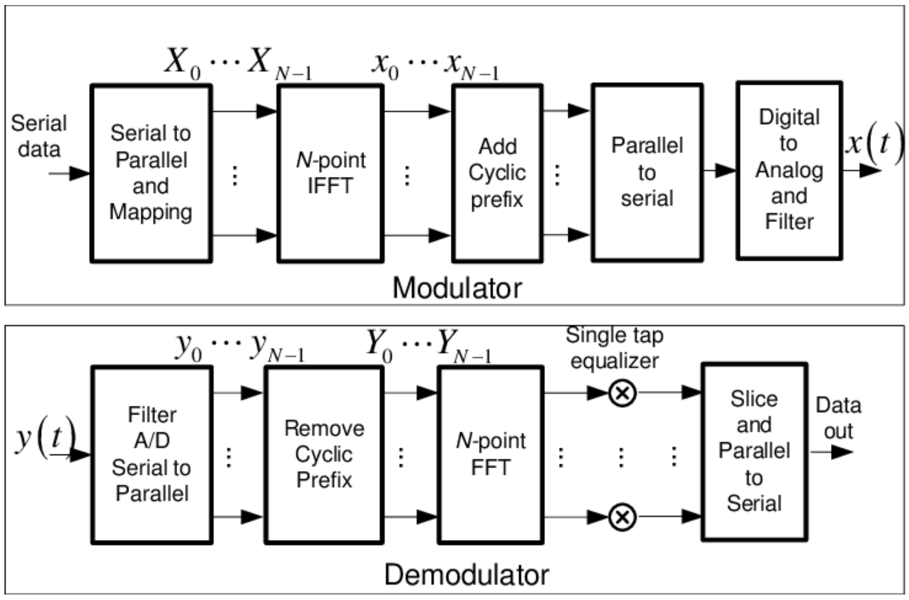


Figure 10: OFDM block diagram for modulation and demodulation (Armstrong et al. 2006)

For modulation (upper part of Figure 10), the serial input datastream is first divided into N number of smaller blocks, and each block is mapped to a set of complex symbols $X[0], X[1], ..., X[N-1]$, using a chosen modulation scheme (e.g. MQAM). These modulated symbols are then arranged in a parallel fashion, forming an OFDM symbol - which are the N discrete frequency components seen in Figure 9. Each of N parallel OFDM symbols are converted from frequency to time domain using an inverse fast Fourier transform (IFFT) operation. This step effectively modulates the data onto orthogonal subcarrier time sequence $x[n] = x[0], ..., x[N-1]$ of length N, given by

$$x[n] = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} X[i] e^{(j2\pi ni)/N}, \quad 0 \leq n \leq N-1, \tag{31}$$

where $x[n]$ is the OFDM symbol sample set.

To mitigate ISI caused by multipath propagation, a cyclic prefix (CP) is added to each OFDM symbol by copying the last few samples of the symbol and appending them to the beginning (Figure 11). Finally, the OFDM symbols with the added CP are then rearranged back to a serial stream for transmission over the channel. In addition, the signal needs to be made analogous and filtered, but that happens on the hardware side. (Prasad 2004)

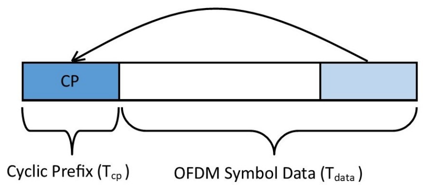


Figure 11: Cyclic prefix (CP) is added for ISI rejection

The process of demodulation is in essence the reverse of the OFDM modulator (lower part of Figure 10). It is important to note the removal of cyclic prefixes and use of FFT (the opposite of what is performed in modulation).

3 Methodology

3.1 Expanding the cost function

One of the goals in this project was to expand on an existing cost function. The cost function and its constraints comes from (Iadarola 2022). The cost function represents the bit rate per packet, meaning the average bit rate over a certain time period, if one were to constantly send packets in this period. This function takes into account the time it takes to encode and send the packet, and is written as:

$$R_p = \frac{mR_cB(N - n_x)\log_2(M)}{m(1 + p_c)N + B(t_{oh} + t_d)} \tag{32}$$

To make this function into a convex function to minimize instead, the logarithm is taken and it is multiplied by -1. This is already shown for the old cost function in the thesis it came from as becoming:

$$-J_0 = \log m + \log R_c + \log B + \log(N - n_x) + \log(\log_2(M)) - \log(m(1 + pc)N + B(t_{oh} + t_d)) \tag{33}$$

Because of the amount of variables, an overview of the different variables used in the cost function and constraints are given in table 2, which is based on the same thesis the cost function comes from. Most of the variables in the table will not be discussed in this report as they're not the focus. Unless otherwise specified, the provided value of a variable is not based on anything and is just there to have a baseline to simulate with.

The modification is done in two parts; one is changing the measure of the cost that is being minimized. The other is more accurately estimating SINR by optimizing transmitted power and including a model for noise and absorption.

Variable	Meaning	Type	value
m	Symbols per packet	Optimizable	
N	Subcarriers	Optimizable	
M	Modulation order	Optimizable	
P	Transmit power	Optimizable	
p_c	Cyclic prefix fraction	Estimated	0.25
τ	Delay spread	Estimated	0.025
ν	Doppler spread	Estimated	1
P_n	Noise power	Estimated	see section 3.1.2
p_b	BER	Estimated	see equation 36
r	Range	Estimated	300
t_{oh}	Overhead	Estimated	0.001
B	Bandwidth	Known	4000
L	Energy spread threshold	Known	0.01 see section 3.1.2
c	Speed of sound	Known	1500
$P_{\min}$	Min TX power	Device specific	-1.87 see section 3.1.2
$P_{\max}$	Max TX power	Device specific	18.13 see section 3.1.2
k	Relative doppler margin	Design	2
p_l^t	Target packet loss	Design	0.1
n_x	Non-data subcarriers	Design	0
R_c	Coding rate	Design	0.5
p_l	Packet loss ratio	Derived	see equation 37

Table 2: Variables

The new cost function, which will be called $R_s(x)$ is a function of the optimizable variables:

$$x = \begin{matrix} m \\ N \\ M \\ P \end{matrix}$$

$$R_s(x) \text{ is subject to } \begin{cases} m & \geq & 1 \\ m & \leq & 40 \\ N & \geq & \tau \frac{B}{p_c} \\ N & \leq & \frac{B}{k\nu} \\ M & \geq & 2 \\ M & \leq & 64 \\ P & \geq & P_{min} \\ P & \leq & P_{max} \\ p_l & \leq & p_l^t \end{cases} \tag{34}$$

Each of the inequality constraints was applied to the cost function via the barrier function. An example of how this looks using the two constraints of m:

$$R_s(x) - \frac{1}{t}(\log(m-1) + \log(40-m))$$

3.1.1 Optimizing for successful bit rate per packet

Because a measure of just bit rate doesn't tell much about how many bits are actually received, a modification was done with the hope that it would improve the number of bits that are received. This new cost is for the *successful bit rate per packet.* A visual way to think of this change can be found by comparing figure 12 and figure 13 (the numbers are just made up to illustrate the point). Figure 12 illustrates the old "best parameters" that has managed to achieve a bit rate per packet of 1 kbps, but due to loss, only an average of 600 bps are actually received. This modification aims to achieve a scenario as shown in figure 13. In this scenario the bit rate per packet is only 900 bps, but the amount that reaches the receiver has increased to 800 bps, meaning the performance has increased.

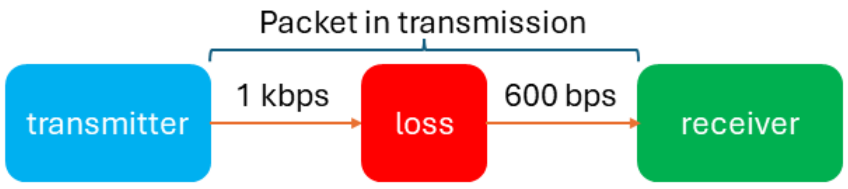


Figure 12: Situation with high bit rate, but low successful bit rate

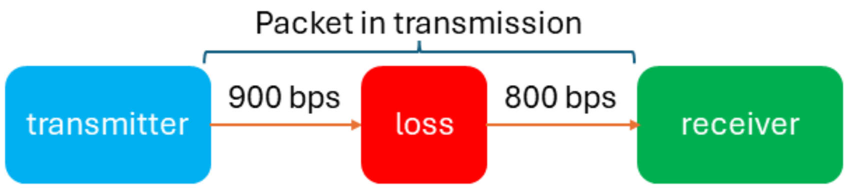


Figure 13: Situation with lower bit rate, but higher successful bit rate

The way this modification was done was by scaling the value of the old cost function $\frac{bits}{packet}$ with the chance that the packet is not lost. It is assumed that any bit errors will cause the packet to be lost (Iadarola 2022). The average chance of a bit error is defined using the definition of average BER from (Goldsmith 2005):

$$\text{BER} = \frac{\text{bits}_{\text{error}}}{\text{bits}_{\text{total}}} \tag{35}$$

BER varies depending on the modulation technique used (Goldsmith 2005). For this project it is assumed that MQAM (2.6.1) is used. (Goldsmith and Chua 1997) approximates the BER for a single-carrier MQAM as:

$$p_b \approx 0.2 \exp\left(-\frac{3}{2(M-1)}\overline{\gamma}\right) \tag{36}$$

Given that $0 < \overline{\gamma}^{\mathrm{dB}} < 30$ and $M \geq 4$, where $\overline{\gamma}$ is the average SNR (in this case SINR) at the receiver. The chance of a packet being lost p_l is derived in (Iadarola 2022) as:

$$p_l = 1 - \left(1 - p_b^{1/R_c}\right)^{bRc} \tag{37}$$

Where b is coded bits per packet $b = m(N - n_x)\log_2(M)$. Because the chance of a packet *not* being lost is just $1 - p_l$, the chance of a successful package is:

$$s_p = \left(1 - p_b^{1/R_c}\right)^{bR_c} \tag{38}$$

Scaling the bit rate per packet R_p with s_p yields the successful bit rate per packet. Because multiplying becomes addition when dealing with logarithmic values, the new cost function can be found by simply subtracting the log of s_p from $-J_0$:

$$R_s = -J_0 - \log(s_p) = -J_0 - bR_c \log\left(1 - p_b^{1/R_c}\right) \tag{39}$$

3.1.2 Improving SINR

From (Iadarola 2022), the average SINR is calculated as:

$$\overline{\gamma} = \frac{P_{rx}}{LP_{rx} + P_n} \tag{40}$$

L is a factor of the amount of interference, and it will be assumed to be 0.01, meaning the SINR cannot go above 100. This is also to ensure that $0 < \overline{\gamma}^{\mathrm{dB}} < 30$ holds; in this scenario $\max\left(\overline{\gamma}^{\mathrm{dB}}\right) = 10\log_{10} 100 = 20$ dB. P_n is the noise power and P_{rx} is the received power; both are measured in watts.

To find P_n, the equations from section 2.3.4 were used with some assumptions. Because OFDM has subcarriers at different frequencies, each subcarrier is experiencing a different noise level. It was assumed that the difference was small enough that the colored noise can be approximated to white noise with a power spectral density that corresponds to the NSL value seen in the Wenz-curve at the median frequency of the operating range of the modems:

$$\mathbf{f} = \frac{18 + 34}{2} = 26 \text{ kHz} \tag{41}$$

To get a somewhat noisy environment, shipping density was set to 0.5, and wind speed to 10 m/s. Looking at figure 3, the noise in this frequency is dominated by thermal noise and surface agitation, and the NSL value is about 45 dB re 1 μPa/Hz. NL is gained by using formula from 25.

For this to be used to measure SINR, the noise level has to be converted to a noise power P_n which expresses the noise in watts. This can be done by using the formula for source level in section 2.3.3, and solving for P:

$$P_n = 10^{(\mathrm{NSL} + 10\log(B) - 170.8)/10} \tag{42}$$

To find the absorption coefficient α, the model from section 2.3.6 was used, with the assumption that the temperature and depth stays the same, and with the same median frequency that was used for calculating noise. Because the coefficient is per km, it is scaled with 1/1000 to get it in meters instead.

From (Iadarola 2022), the received power was calculated as $P_{rx}^{\text{dB}} = P_{tx}^{\text{dB}} - 20\log_{10}(r) - \alpha(\mathbf{f})r$. r is the range between transmitter and receiver, P_{tx} is the source level of the transmitter, and α is the absorption coefficient of the medium. Because the modems used for this project transmit omnidirectionally, P_{tx} can be described with the source level representation for spherical spreading $P_{tx} = P + 170.8$. With the current information the received power can be calculated as:

$$P_{rx}^{\text{dB}} = P^{\text{dB}} + 170.8 - 20\log_{10}(r) - \alpha(\mathbf{f})r/1000 \tag{43}$$

Because this level needs to be in watts, the calculation for source level was used, which means the reference value is subtracted before converting from decibels:

$$P_{rx} = 10^{\left(P^{\text{dB}} - 20\log_{10}(r) - \alpha(\mathbf{f})r/1000\right)/10} \tag{44}$$

By inserting equation 44 and 42 into 40, it was possible to make transmission power an optimizable variable.

$P_{\min}$ and $P_{\max}$ was found by using the documentation of the modems used. From (EvoLogics 2018) the maximum power that can be transmitted is 65 W, making $P_{\max} = 10\log_{10}(65) = 18.13$ dB. (EvoLogics 2021) has the different source levels the modem can use, with the minimum being -20 dB of max. The minimum power was thus calculated as $P_{\max} - 20 = 18.13 - 20 = -1.87$ dB.

3.1.3 Simulating with the new cost function

The simulator used to test the new cost function is a modified version of the one used in (Iadarola 2022). Some unused functions and variables have been removed so that almost only code that was actually used remained. Most of the work done with the cost function is theoretical work, and therefore the simulator codebase is very similar to how it originally was, with the exception of *SimulationFunctionXTX_BTX.py*, which has been completely revamped.

When simulating without optimizing P, SINR is set to a constant value of 100. All simulations were simulated with environmental parameters being 12°C, depth of 30 meters, and 300 meters distance between each of the modems. The simulations used simulated asynchronicity (explained in more detail in (Iadarola 2022)) and 10 percent chance of packet loss. Experimentation based on what led to best stability and accuracy vs simulation time led to a chosen step size of 0.005 and a t value of 10 for the barrier function. Although the code came with IPM implemented, it was not used as the extra accuracy was not deemed important.

3.2 Equipment list

3.2.1 Hardware

Name	**Quantity**	**Description**	**Manufacturer**
S2C R 18/34 Hydroacoustic Modem w/USBL	2	Acoustic modem	EvoLogics
Data/Power 6 pin SubConn cable	2	Female SubConn cable with 3 power connectors and 1 Cat.6A ethernet cable	EvoLogics
5-Port Gigabit Ethernet Unmanaged Switch	1	Ethernet switch	Gigabit
Cat.5e ethernet cable	2	Ethernet cable for connecting computers to the Gigabit switch	Teltonika
Tenma 72-6615	1	Laboratory power supply with two independently adjustable power supplies	Tenma
Tattu Plus 12000mAh 6S 22.2V 15C Lipo Smart Battery Pack	2	Battery for powering the modems in the field	Tattu Plus
Laptop	2	Laptop for connecting to acoustic modems via ethernet cable.	Any

3.2.2 Software

Name	**Version**	**Description**	**Manufacturer**
Ubuntu	22.04	Operating system for the laptops	Canonical
Python	3.10.12	Python standard libraries, interpreter and compiler	Python Software Foundation
GNU Compiler Collection	Ubuntu 11.04.0	Compiler for ”C”	Free Software Foundation
EvoLogics AMA	2.0.5	GUI software for testing acoustic modem communication	EvoLogics
Janus Tool Kit	3.0.5	Tool Kit with samples and libraries for JANUS encoding / decoding	NATO CMRE
sdmsh		Libraries and command-line tools for commucation between EvoLogics acoustic modems	EvoLogics

Additional software requirements in the form of Ubuntu tools and Python libraries will be described in Methodology.

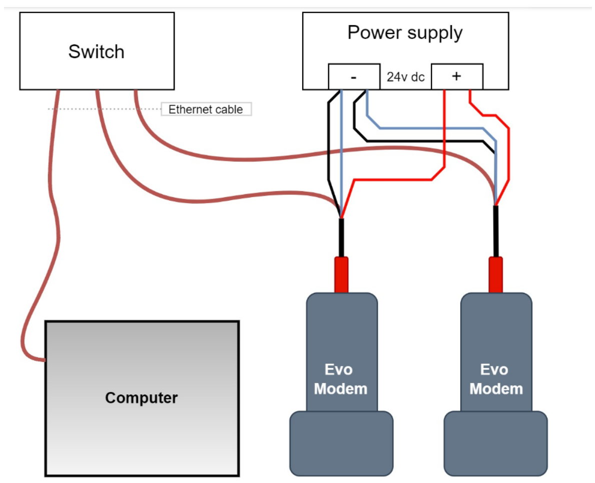


Figure 14: A example diagram of how the EvoLogics modem are to be connected ('Distributed Optimization Based Adaptive Underwater Communication Schemes' 2023).

3.3 Physical setup

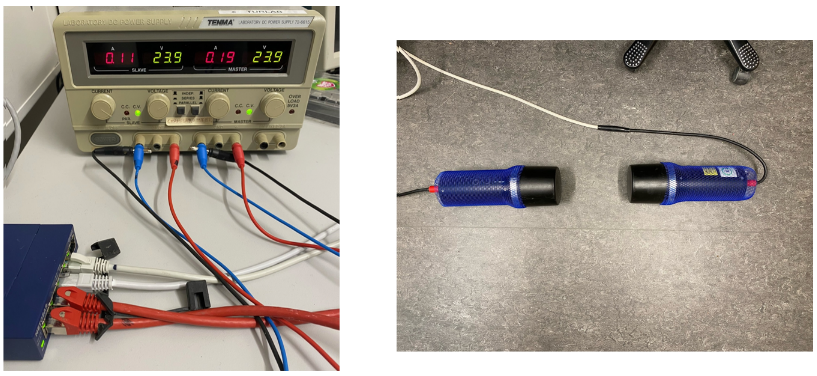


Figure 15: How the EvoLogics modems were connected and setup for most of the air testing

3.4 Python as an interface

As mentioned in Theory, a Python library is made using SWIG, interfacing Python with the EvoLogics modems. We refer to the library as *sdm*, while the functionality is stored in scripts called *sdmapi.py* and *sdm.py*. Through *sdm* we gain access to some key functions. One of them is

```
create_session(name, addr)
```

This creates a session with a modem, given that *addr* is the modems IP-address. This will kick out any other user that is connected to the modem and flush the connection. After a session is initialized

```
send_stop(session)
```

will ready the modems to receive further commands. A *send_stop()* will put the modem in Software Defined Modem mode and stop any ongoing commands. This function is often used as a way to reset the modems between commands, and proves useful if the modems at any point get hung up and freeze. To transmit data;

```
send_signal_file(session. ref_filename, signal_filename)
```

This function does two important things; firstly it concatenates *ref_filename* and *signal_filename* into a single signal stream - where *ref_filename* is the preamble. and *signal_filename* is the data to be transmitted. After concatenating it transmits the signal data in one continuous transmission. Before receiving transmissions, a buffer to store the incoming signal data must be initialized. This is done by

```
add_sink_membuf(session)
add_sink(session, path)
```

This will initialize a memory buffer to be filled on the modem. *add_sink()* will specify which file in which path you want to store the filled buffer after reception is complete. Now one can begin to receive data by

```
send_rx(session, nsamples)
get_membuf(session)
```

In *send_rx()* the user will specify how many samples one wishes to listen for. The number of samples selected will affect the size of the output file defined in *add_sink()*. *get_membuf()* will save the filled buffer in the path specified in *add_sink()*.

An additional function, which is useful as a debugging tool is

```
expect(session, args)
```

After issuing a command, also running a *expect()* will report if the command was completed successfully, and it will repeat the parameters of the command back to you. This works as a live monitoring tool during execution.

Using the functions described above in a Python script allows continuous communication without having to run terminal commands for every transmission and reception. All these functions will be utilized in the execution of the ra-NRC algorithm, for which the methodology will be discussed later in the thesis.

3.5 JANUS implementation

As described in Theory, NATO has released an implementation of JANUS to the public, called JANUS Tool Kit 3.0.5. This implementation makes packet generation using JANUS easy, and to use the tool kit one does not need a substantial knowledge of the theory behind JANUS, as it gets handled by a "black box". Packet generation and decoding using the JANUS toolkit is done by the *janus-tx/rx* executables, where config files are the parameters. The config files include a list of several mandatory and optional parameters:

```
--pset-file
--pset-id
```

–pset-file will be a table (.csv format) of different center frequencies and bandwith for the signal generation. *–pset-id* will select which entry in the table you want to base your signal generation on. The Evologics 18/34 modem, with its own central frequency and bandwith, is entry number 2 in the *parameter_sets.csv* used as a table in this thesis, thus *–pset-id* will also be 2.

```
--stream-driver
--stream-driver-args
```

–stream-driver and *–stream-driver-args* select the media format of the input / output stream, e.g .raw or .wav or tcp (TCP). Different media formats may require different additional arguments. The sampling rate of the stream is defined by

```
--stream-fs
```

and should satisfy the Nyquist criteria of twice the sample rate of the highest frequency in the signal. In *parameter_sets.csv*, the maximum frequency is set to be 31 360 Hz, to make sure we satisfy the Nyquist criteria we set the stream sampling requency to 96 000 Hz. Using a higher sampling frequency than this lead to too much data having to be transferred, as the waveform file would increase in size. The format of each sample is described by

```
--stream-format
```

where you have the option to store each sample as anything from a 8 bit Pulse Code Modulation (PCM) to a 32 bit PCM. We opted for a 16 bit representation as it proved the most reliable.

A transmission-specific argument is

```
--packet-cargo
```

which is the string to be encoded into the cargo, and will function as the data one wishes to transmit, in addition to the headers. Lastly, for debugging one may wish to include

```
--verbose
```

where you can decide between 0-9. 9 will report every packet received in great detail, 0 will report nothing.

3.5.1 janus-tx

The NRC-algorithm requires the exchange of one integer and 20 floating point values. Encoding these values into a JANUS packet and one can expect a packet size of anywhere between 400 000 and 1 200 000 samples. A sample in this case is represented as signed 16 bit integer of Pulse Code Modulation. During development there was discovered a non-linearity in the bit-error-rate one would encounter based on JANUS packet size. For packets with a size of less than 400 000 samples, one could expect a bit-error rate of less than 0.05, however, were one to double the amount of samples to 800 000, the bit-error rate would increase to 0.25. The cause for this non-linear increase in bit-error rate is unidentified but the fault likely lies within the *sdmsh*-implementation, and not the JANUS implementation. Sending packets with a size of 1 200 000 samples in a single transmission would result in a high bit-error rate thus making the payload undecipherable. All transmissions were therefore to be split across two JANUS packets, as sending two packets with a size of 600 000 samples is more reliable than sending a single packet of 1 200 000 samples.

To initialize JANUS packet generation one must run the following commands in two separate shells:

```
./janus-loop --config-file payload.ini
```

and

```
./janus-loop2 --config-file payload2.ini
```

, using the following configurations of *payload.ini* and *payload2.ini*:

```
--pset-file ./etc/parameter_sets.csv
--pset-id 2
--stream-driver raw
--stream-driver-args payload1.raw
--packet-cargo ""
--stream-fs 96000
--stream-format S16
--verbose 9
```

```
--pset-file ./etc/parameter_sets.csv
--pset-id 2
--stream-driver raw
--stream-driver-args payload2.raw
--packet-cargo ""
--stream-fs 96000
--stream-format S16
--verbose 9
```

These commands execute two "C" scripts, which are a modified version of the *janus-tx* from the JANUS Tool Kit. One key difference is they are made to continuously generate packages using a while()-loop. The original *janus-tx* only permitted a single packet generated from a single command line argument. The modified version instead reads a shared resource *shared_string.txt* into a buffer:

```
FILE *fp = fopen("shared_string.txt", "r");
   if (fp != NULL){
     fgets(buffer, 512, fp);
     fclose(fp);
```

and then uses half of that buffer in packet generation:

```
int half_length = strlen(buffer) / 2;
     strncpy(cli_options->opts[PACKET_CARGO].arg, buffer, half_length);
```

janus-loop and *janus-loop2* will use each half of the buffer respectively. Now any program may edit the contents of *shared_string.txt* and two JANUS packages will automatically be generated with the payload spread across them, stored as *payload1.raw* and *payload2.raw*.

3.5.2 janus-rx

In order to receive JANUS packets, the JANUS receiver has to be initialized from the command line using:

```
./janus-rx --config-file rxcfg.ini
```

using the following configuration in *rxcfg.ini*:

```
--pset-file ./etc/parameter_sets.csv
--pset-id 2
--stream-driver tcp
--stream-driver-args listen:127.0.0.1:9998
--stream-fs 96000
--stream-format S16
--verbose 1
```

This executes a "C" script, which is a modified version of *janus-rx* from the JANUS Tool Kit. Where this script differs from the original is it accounts for the dual-packet generation from the tx-side. First a receiver object is initialized, using the parameters in *rxcfg.ini*:

```
simple_rx = janus_simple_rx_new(params);
```

Then carrier sensing is initialized, as per *–stream-format S16*, it tries to decode the incoming stream as int_16t's:

```
carrier_sensing = janus_carrier_sensing_new(janus_simple_rx_get_rx(simple_rx));
```

and upon detection, a packet object is initialized:

```
packet = janus_packet_new(params->verbose);
```

In order to account for the aforementioned dual-packet generation, some boolean logic has been implemented into the script.

```
rx_mode = (rx_mode == 1) ? 2 : 1;
```

janus-rx is a looping script, each iteration *rx_mode* will flip between 1 and 2. *rx_mode* will then be passed as an argument into:

```
janus_packet_dump(packet, rx_mode);
```

and *janus_packet_dump()* has been modified to store the first and second packet separately:

```
 if (rx == 1){
         FILE *f = fopen("cargo_data1.txt", "w");
         if (f == NULL)
         {
             printf("Error opening file!\n");
```

```
            exit(1);
        }
        fprintf(f, "%s", string_cargo);
        fclose(f);
      }
  if (rx == 2){
        FILE *f = fopen("cargo_data2.txt", "w");
        if (f == NULL)
        {
            printf("Error opening file!\n");
            exit(1);
        }
        fprintf(f, "%s", string_cargo);
        fclose(f);
      }
```

string_cargo is the character array which represents the decoded cargo of the packet. For the first packet received it is stored in *cargo_data1.txt* and for the second packet it is stored in *cargo_data2.txt*. *janus-rx* will continue receiving packets infinitely and flip between storing the cargo in the two *cargo_data.txt* files.

3.6 ra-NRC

The code used for ra-NRC in this project is based on the simulation code used in (Iadarola 2022). All simulation specific code has been removed, such that only the code that handles the ra-NRC itself remains left behind, though it has been built upon to fit an actual physical system. The parts of methodology here that concerns doing anything from the ra-NRC is thus not descriptive of us converting the pseudocode from (Bof et al. 2019) into Python code, but rather a description of what was already there, and how we have worked with the current implementation to fit our project.

ra-NRC_main.py is the program that is run in order to start the consensus algorithm. It is responsible for running the four main steps of the algorithm: **Initialization**, **Data Transmission**, **Data Reception** and **Estimate Update**. All calculated values and what happens inside each main step is stored in and handled by the *Node* class in *Node.py*.

Before running the main program, the modems must be initialized using the terminal. The modems are connected to using *sdmsh*. Each modem is then configured to use a *preamble.raw* as a reference signal. Depending on the conditions of the test one wants to conduct, a pre-amplifier gain and source level must also be set. For air testing the best configuration proved to be 40 for pre-amplifier gain and source level 2. This only has to be done once, which is after connecting the modems to power. After initializing the modems the *janus-tx* and *janus-rx* has to be initialized as well, as described in Section 3.5.

When *ra-NRC_main.py* starts, required consensus algorithm parameters and node specific values are set. The node specifics are hard coded, and thus needs to be changed manually for each node depending on the network setup:

```
# node specific
node_id = 1
neighboring_nodes = np.array([1]) # ID list

# parameters
epsilon = 0.005
c = 0.00000001
MAX_ITER = 500
IPM_cycle = 1
bb = 10
x0 = np.array([22, 1250, 14, 28])  # initial point
```

Then some communication parameters are set before the program actually begins doing the ra-NRC algorithm. The first step of the algorithm is initializing the values. The pseudocode for this is:

Initialization

```
1: $x_i \leftarrow x^0$
2: $y_i \leftarrow 0, g_i \leftarrow 0, g_i^{\text{old}} \leftarrow 0$
3: $z_i \leftarrow I_n, h_i \leftarrow I_n, h_i^{\text{old}} \leftarrow I_n$
4: $\sigma_{i,y} \leftarrow 0, \sigma_{i,z} \leftarrow 0$
5: $\rho_{i,y}^{(j)} \leftarrow 0, \rho_{i,z}^{(j)} \leftarrow 0 \quad \forall j \in \mathcal{N}_i^{\text{in}}$
6: $\text{flag}_{\text{reception},i} \leftarrow 0, \text{flag}_{\text{update},i} \leftarrow 0$
7: $\text{flag}_{\text{transmission},i} \leftarrow 1$
```

In code, this is handled by initializing the *Node* object which resets all values to their initialization state. The function then enters the main loop where it runs *node.transmit_data*, *node.receive_data* and *node.update_estimation* until the set number of iterations. The code for using the IPM has been left intact, which means that in theory the program can do several IPM-cycles, though this has been neither tested nor tuned. When the loop has completed, the program just exits normally. The resulting parameters can be found by calling *node.xi*.

3.6.1 Update estimation

This method is responsible for calculating the next x_i and updating h_i and g_i. First it is checked whether the eigenvalues of z_i is smaller than that of cI; if it is then z_i is set to cI. This is to ensure that the basin of attraction is big enough so that the values converge. Then the next x_i is calculated. The pseudocode for the rest is:

Estimate Update

```
1: $x_i \leftarrow (1-\varepsilon)x_i + \varepsilon z_i^{-1} y_i$
2: $g_i^{\text{old}} \leftarrow g_i$
3: $h_i^{\text{old}} \leftarrow h_i$
4: $h_i \leftarrow \nabla^2 f_i(x_i)$
5: $g_i \leftarrow h_i x_i - \nabla f_i(x_i)$
6: $y_i \leftarrow y_i + g_i - g_i^{\text{old}}$
7: $z_i \leftarrow z_i + h_i - h_i^{\text{old}}$
8: $\text{flag}_{\text{update},i} \leftarrow 0$
9: $\text{flag}_{\text{transmission},i} \leftarrow 1$ (optional)
```

The code implementation is pretty straightforward, though there is some matrix calculations. To get the gradient and Hessian, the method *jacobian* from the python library *autograd* was used:

```python
self.xi = (1 - self.epsilon) * self.xi +
          np.matmul((self.epsilon * np.linalg.inv(self.zi)), np.transpose(self.yi))

self.gi_old = self.gi
self.hi_old = self.hi

self.hi = jacobian(jacobian(self.cfn.get_fn))(self.xi, bb=self.bb)

self.gi = np.subtract(np.matmul(self.hi, self.xi.transpose()),
                      jacobian(self.cfn.get_fn)(self.xi, bb=self.bb))

self.yi = self.yi + self.gi - self.gi_old
self.zi = self.zi + self.hi - self.hi_old
```

3.6.2 Transmission logic

Transmission is the first thing that happens after a node is initialized. It works by invoking a method from the *Node* class:

```python
    node.transmit_data()
```

The pseudocode to the ra-NRC for this part is:

Data Transmission

```
1: transmitter_node_ID ← $i$
2: $y_i \leftarrow \frac{1}{|\mathcal{N}_i^{\text{out}}|+1} y_i$
3: $z_i \leftarrow \frac{1}{|\mathcal{N}_i^{\text{out}}|+1} z_i$
4: $\sigma_{i,y} \leftarrow \sigma_{i,y} + y_i$
5: $\sigma_{i,z} \leftarrow \sigma_{i,z} + z_i$
6: Broadcast: transmitter_node_ID, $\sigma_{i,y}$, $\sigma_{i,z}$
7: $\text{flag}_{\text{transmission},i} \leftarrow 0$
```

Three variables are to be transmitted between nodes, they are *Node_ID*, $\sigma_{i,y}$, and $\sigma_{i,z}$. Data transmission is responsible for downscaling y_i and z_i, serializing all variables to be transmitted into one string, wait for JANUS to encode the string into two packets, then transmit the packets using *sdm*.

In *transmit_data()*, $\sigma_{i,y}$, and $\sigma_{i,z}$ first get downscaled according to the amount of neighboring nodes. *Node_ID* is an integer and $\sigma_{i,y}$ is an array, thus they are already of dimension one and easy to serialize. $\sigma_{i,z}$ however is a 4x4 matrix. To serialize it, it gets flattened into an array of 16 values. The variables then get converted into strings and appended to each other, but separated by a set of square brackets "[ ]". An example string might look like:

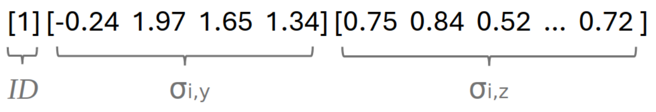


Figure 16: An example of the string to be transmitted. An additional 12 values would exist between the dots of $\sigma_{i,z}$

This string is then written to *shared_string.txt* where the already running JANUS implementation will automatically generate two packets from it. As the JANUS implementation runs on separate scripts (written in "C"), there is no synchronization with *NRC_main.py*. Before *NRC_main.py* moves forward - a sleep function of one second gets invoked, to ensure that the JANUS implementation has had sufficient time to generate two packets. The two packets will be stored as *payload1.raw* and *payload2.raw* in *payload_path*

After waiting for a second, the first package gets sent using *sdm*:

```
    sdm.send_stop(self.session)
    sdm.expect(self.session, sdm.REPLY_STOP)
    sdm.send_signal_file(self.session, preamble_path, payload_path)
```

sdm.send_stop() clears the buffers of the modem, and expects a reply to ensure the modem is ready. *sdm.send_signal_file()* appends the preamble to the packet and then transmits using the modem. After transmitting the first package, the script waits for 0.5 seconds before transmitting the second package, using the same method. The waiting period of 0.5 seconds is to ensure that there are no residual echos that may interfere with packet number two.

While doing a lot of transmissions with the modems, sometimes the modems would report a "DROP"-error and freeze mid-transmission. This error was mostly fixed by ensuring a *send_stop()* between transmissions, as the error was likely due to an overfilled buffer. However, on odd occasions the modems would still freeze mid-transmission. To ensure that it would not remain frozen permanently, a watchdog timer is running on a separate thread that breaks the transmission logic to unfreeze the modem, if it gets stuck for more than 10 seconds.

3.6.3 Reception logic

After transmitting, the node will enter the reception state. The pseudocode for the ra-NRC for this part is:

Data Reception

```
1: $j \leftarrow$ transmitter_node_ID,    $(j \in \mathcal{N}_i^{\text{in}})$
2: $y_i \leftarrow y_i + \sigma_{j,y} - \rho_{i,y}^{(j)}$
3: $z_i \leftarrow z_i + \sigma_{j,z} - \rho_{i,z}^{(j)}$
4: $\rho_{i,y}^{(j)} \leftarrow \sigma_{j,y}$
5: $\rho_{i,z}^{(j)} \leftarrow \sigma_{j,z}$
6: $\text{flag}_{\text{reception},i} \leftarrow 0$
7: $\text{flag}_{\text{update},i} \leftarrow 1$ (optional)
```

Three variables are to be received from a neighbor. *Node_ID*, $\sigma_{j,y}$, and $\sigma_{j,z}$. A node begins listening by invoking a method from the *Node* class:

```
    ID, sigma_yj, sigma_zj = node.receive_data(listen_time)
```

receive_data() will use *sdm* to receive packets:

```
    sdm.send_stop(self.session)
    sdm.expect(self.session, sdm.REPLY_STOP)
    sdm.add_sink_membuf(self.session)
    sdm.add_sink(self.session, "rx.raw")
    sdm.send_rx(self.session, 2000000)
    sdm.expect(self.session, sdm.REPLY_STOP)
```

send_stop() will clear any buffers, and *add_sink_membuf()* will ready a new buffer. *add_sink()* will set the destination file for the stream to *rx.raw*. *send_rx()* puts the modem into listening mode, where it will start filling the buffer as soon as it hears a preamble. The buffer will be filled until it reaches 2 000 000 samples. It is set to 2 000 000 as we expect to receive two packets, which may have a size of 1 600 000 samples in total, in addition to some waiting between packets, which will also fill up the buffer.

sdm doesn't provide the ability listen for X amount of seconds, listen time is only defined by number of samples. Only after hearing a preamble will the modem start filling the buffer with samples, so if it doesn't hear any preambles - it will be stuck listening forever. To not get stuck like this, a watchdog timer is running on a separate thread that will break the reception logic after *listen_time* amount of seconds. *listen_time* is defined in *NRC_main.py*.

After the buffer has been filled and saved to *rx.raw*, it will sent to the JANUS decoder, which is running as a separate script in the background.

```
    data = np.fromfile(file_path, dtype=np.uint16)
    data_bytes = data.tobytes()
    self.socket.sendall(data_bytes)
```

The raw bit data of *rx.raw* is read into a NumPy array as int16's. It is then converted to bytes using *tobytes()*, and sent over 127.0.0.1:9998 using *socket.sendall()*. As explained in the *janus-rx* logic, the JANUS receiver will be listening for a stream on 127.0.0.1:9998, and try to decode it as int16's. If the stream contains a valid JANUS signal, the JANUS receiver will write the payload of the received packets into *cargo_data1.txt* and *cargo_data2.txt*. As mentioned in the transmission logic, there is no synchronization between *NRC_main.py* and the JANUS "C" implementations. Thus *NRC_main.py* will sleep for one second to ensure that *janus-rx* has time to decode the packets.

After *janus-rx* has decoded the packets, *NRC_main.py* will merge the contents of *cargo_data1.txt* and *cargo_data2.txt*, to reconstruct the string generated by the senders transmission logic. Ideally the reconstructed string will look like Figure 16. *Node ID*, $\sigma_{j,y}$, and $\sigma_{j,z}$ will be extracted from the string by converting the contents of the first bracket to an integer, and the contents of the next two brackets as floats. Due to the nature of acoustics, packets will often appear corrupted, and the corresponding cargo will contain gibberish. Sometimes the entire cargo will be gibberish, other times only a few values are corrupted. In both cases, to ensure this gibberish doesn't get turned into variables, checks get run on whether the cargo string contain the expected format and number of values. Any variation from the expected format will raise an exception and exit the reception logic, resulting in a packet loss.

As mentioned earlier, $\sigma_{j,z}$ represents a 4x4 matrix, but is currently an array of 16 values. Before all variables get returned , $\sigma_{j,z}$ gets reshaped back into a 4x4 matrix. Now *Node ID*, $\sigma_{j,y}$, and $\sigma_{j,z}$ get returned back into the main loop.

Upon receiving a new set of variables; *Node ID*, $\sigma_{j,y}$, and $\sigma_{j,z}$, they get compared to the previous set of received variables. If they vary too much, a bit error might have occurred which has flipped a single digit without raising an error in decoding. If they vary too much they get discarded, similar to how a digital low pass filter works.

The main program flow is shown visually in Figure 17.

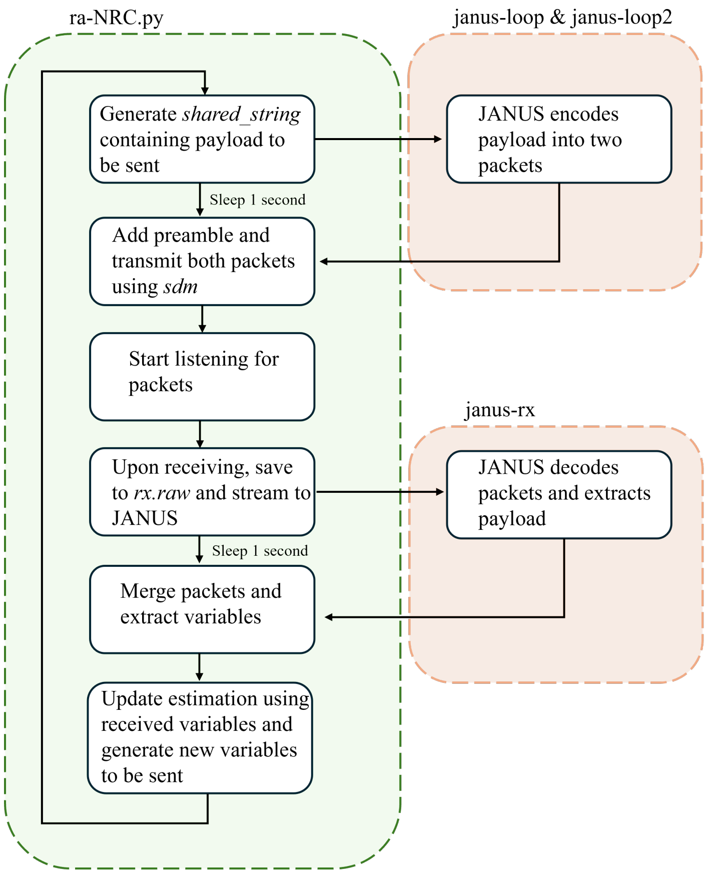

ra-NRC.py
janus-loop & janus-loop2
Generate *shared_string* containing payload to be sent
JANUS encodes payload into two packets
Sleep 1 second
Add preamble and transmit both packets using *sdm*
Start listening for packets
janus-rx
Upon receiving, save to *rx.raw* and stream to JANUS
JANUS decodes packets and extracts payload
Sleep 1 second
Merge packets and extract variables
Update estimation using received variables and generate new variables to be sent

Figure 17: The main program flow of ra-NRC_main.py, excluding termination logic

3.7 OFDM implementation

OFDM is a widely used communication algorithm, having a presence in modern technologies such as the Wi-Fi IEEE 802.11 standard, and WiMAX IEEE 802.16, amongst others (Goldsmith 2005). There is also abundant research for it in UWAC applications (Stojanovic 2006, Li et al. 2009, Radosevic et al. 2014).

The OFDM system used in this project is based on initial implementation provided by Zhengnang Li and Milica Stojanovic. All of the OFDM logic is contained within the file *OFDM.py* within the folder *ofdm*.

3.7.1 Preamble generation and detection

The preamble is a known sequence of symbols transmitted at the beginning of each OFDM frame. It is essential to have it to be able to synchronize data endpoints between the transmitting and receiving nodes in the system. It allows the receiving side to accurately identify the start of transmission. The preamble is generated using a deterministic algorithm with system-consistent parameters, ensuring that all of the agents generate same preamble. The preamble generation is handled by the function *generate_preamble*.

```python
        time_idx, rc_tx = MathOFDM.rrcosfilter(16 * int(1 / self.p_ofdm.B * self.p_ofdm.fs), 0.1, 1 / self.p_ofdm.B, self.p_ofdm.fs)

        sequence = (1+1j)/np.sqrt(2) * (MathOFDM.generate_gold_sequence(self.p_ofdm.n_bits, index=0) * 2 - 1)
        upsampled = np.repeat(sequence, self.p_ofdm.n_sps)
        preamble = sg.convolve(upsampled, rc_tx, mode="same")
        preamble /= np.max(np.abs(preamble))
```

Root-raised-cosinus filter (rrcosfilter) is used to generate a preamble which is efficient for the given available bandwidth.

In function *peak_finder* at the receiver, the preamble is detected using cross correlation between the received signal and the known generated preamble, which helps in identifying the start of the OFDM frame.

```python
        v_preamble = sg.convolve(v_preamble, rc_tx, mode="same")
        xcorr = np.abs(sg.correlate(v_preamble, preamble.conj()))
        xcorr_norm = xcorr / xcorr.max()
        peaks_rx, _ = sg.find_peaks(xcorr_norm, height=0.8, distance=self.p_ofdm.ns_ofdm-1000)
```

3.7.2 Data processing

In function *process_data*, the input data is first divided into smaller blocks, and each block is mapped to a set of complex symbols using a chosen modulation scheme, such as QPSK, MPSK, or MQAM. The modulated symbols are then arranged in a parallel fashion, forming an OFDM symbol.

```python
        bit_diff = self.p_ofdm.n_bits_tx-len(databits)
        if bit_diff < 0:
            raise ValueError(f"Length of passed argument 'databits' cannot exceed maximum available tarnsmission length of {self.p_ofdm.n_bits_tx}. Length of passed 'databits' :{len(databits)}")
        # Add random bits to fill the packet up to "self.p_ofdm.n_bits_tx" length
        bits = np.concatenate((databits, np.random.randint(2, size=(bit_diff))))
        self.bits = bits # for debugging / BER calc

        self.d = None
        if self.modem is None:
            # 4QPSK Modulator
            self.d = MathOFDM.qpskmod_bits(bits)
        else:
            # M-PSK / M-QAM
            self.d = self.modem.modulate(bits)

        self.d = np.concatenate((self.d, np.random.randint(2, size=((self.p_ofdm.K*self.p_ofdm.n_blk)-len(self.d)))))

        self.d = np.reshape(self.d, (self.p_ofdm.K, self.p_ofdm.n_blk))
        # self.d = np.random.choice(self.p_ofdm.alphabet, (self.p_ofdm.K, self.p_ofdm.n_blk))
        data_symbols = np.zeros((self.p_ofdm.K, self.p_ofdm.n_blk), dtype=complex)
        for i_blk in range(self.p_ofdm.n_blk):
            b = np.zeros((self.p_ofdm.K,), dtype=complex)
            b[0] = self.d[0, i_blk]
            for i in range(1, self.p_ofdm.K):
                b[i] = b[i-1] * self.d[i, i_blk]
            data_symbols[:, i_blk] = b
```

3.7.3 Signal generation

Finally for the transmission the OFDM frame is assembled in the function *generate_signal*, where the OFDM symbols are set in series, and an inverse fast Fourier transform (IFFT) is applied to generate the time-domain OFDM signal, as documented in Section 2.6.2. A cyclic prefix (11) is then added to the time-domain signal to combat ISI and maintain orthogonality among subcarriers in the presence of multipath propagation 2.6.2.

```
        u_ofdm = []
        for i_blk in range(data_symbols.shape[1]):
            ifft_modulated = np.fft.ifft(data_symbols[:, i_blk], self.p_ofdm.n_sps
    * self.p_ofdm.K)
            ifft_modulated /= np.max(np.abs(ifft_modulated))

            if self.p_ofdm.is_cp:
                prefix = ifft_modulated[-self.p_ofdm.ns_guard:]
                u_ofdm = np.concatenate((u_ofdm, prefix, ifft_modulated))
            else:
                prefix = np.zeros((self.p_ofdm.ns_guard,))
                u_ofdm = np.concatenate((u_ofdm, ifft_modulated, prefix))
```

The resulting complex symbol sequence $x[n]$ is then made into real-signal of the desired passband frequency using I/Q-modulation method from Figure 2.6.1.

```
            # cos sin for preamble
            cosine_wave_fc = np.cos(2 * np.pi * self.p_ofdm.fc * np.arange(len(
    preamble))/self.p_ofdm.fs)
            sine_wave_fc = np.sin(2 * np.pi * self.p_ofdm.fc * np.arange(len(
    preamble))/self.p_ofdm.fs)

            # cos sin for ofdm
            cosine_wave_f0 = np.cos(2 * np.pi * self.p_ofdm.f0 * np.arange(len(
    u_ofdm))/self.p_ofdm.fs)
            sine_wave_f0 = np.sin(2 * np.pi * self.p_ofdm.f0 * np.arange(len(u_ofdm
    ))/self.p_ofdm.fs)

            # s_modulated = I * cosine_wave - Q * sine_wave  # Note: Q component is
     typically subtracted
            s_preamble = np.real(preamble) * cosine_wave_fc - np.imag(preamble) *
    sine_wave_fc
            s_ofdm = np.real(u_ofdm) * cosine_wave_f0 - np.imag(u_ofdm) *
    sine_wave_f0

            s_preamble /= np.max(np.abs(s_preamble))
            s_ofdm /= np.max(np.abs(s_ofdm))
```

Final signal is then concatenated together from all the different signal components.

```
        s = np.concatenate((np.zeros((int(0.5 * self.p_ofdm.fs),)), # Zeros
                            s_preamble,                              # Preamble
                            np.zeros((self.p_ofdm.ns_pause,)),       # Pause
                            s_ofdm,                                  # Signal
                            np.zeros((self.p_ofdm.ns_pause,)),       # Pause
                            s_preamble,                              # Preamble
                            np.zeros((int(0.5 * self.p_ofdm.fs),))   # Zeros
                            ))
```

Resulting signal s is ready to be sent to a modem for transmission.

3.7.4 Data Estimation

At the receiver in function *estimate_recv_data*, the OFDM signal is first synchronized using the preamble detection process. The cyclic prefix is removed from each OFDM symbol, and a fast Fourier transform (FFT) is applied to convert the time-domain signal back to the frequency domain. The channel estimation obtained from the preamble is used to equalize the received symbols, compensating for the channel distortions. The equalized symbols are then demapped using the corresponding demodulation scheme (e.g., QPSK or QAM) to obtain the original data bits.

```python
        d_hat = np.zeros((self.p_ofdm.K, self.p_ofdm.n_blk), dtype=complex)

        if do_plot:
            plt.figure(figsize=(12, 4))
            plt.subplot(131)

        for i_blk in range(self.p_ofdm.n_blk):
            v_blk = v_ofdm[int(i_blk * self.p_ofdm.Nb) : int((i_blk + 1) * self.p_ofdm.Nb)]

            if self.p_ofdm.is_cp:
                v_to_fft = v_blk[self.p_ofdm.Ng:self.p_ofdm.Nb]
            else:
                v_to_fft = v_blk
                v_to_fft[0:self.p_ofdm.Ng, :] += v_to_fft[self.p_ofdm.Ns : self.p_ofdm.Ns + self.p_ofdm.Ng]
            y_blk_all = self.p_ofdm.Ns * np.fft.fft(v_to_fft, self.p_ofdm.Ns, axis=0)
            y_blk = y_blk_all[0:self.p_ofdm.K]

            h_blk = np.roll(y_blk, 1)
            d_hat[:, i_blk] = h_blk.conj() * y_blk / (h_blk.conj() * h_blk)
```

Final estimated data *d_hat* can then be interpreted according to the expected format of the transmitted data.

3.8 Testing

3.8.1 Air testing

The majority of testing on the modems were done in the lab over air. The setup can be seen in Figure 15. During the initial stages of development, air testing was difficult and unreliable. But after implementing JANUS, and experimenting with different setups, air testing proved to be a reliable method of transmission and reception. The best method proved to be to point the modems directly at each other, separate them by approximately 50 centimeters and use a power level of 2, with a pre-amplifier gain of 40. As we encountered difficulties during early development doing air testing, an early pool-test was conducted.

3.8.2 Pool test

In order to know if the difficulties we encountered were due to the bad transmission qualities of air, we conducted an early pool test. Surprisingly, the results in the pool were significantly worse than air. This was using an early implementation of OFDM encoding, which we were hoping to use for our ra-NRC consensus algorithm. The pool itself was non-ideal for testing, as it was relatively small and the modems would easily pick up reverberations. A diagram of the pool is shown in Figure 18. The results from this test are not discussed in Results, as this test was purely for developmental reasons, and we didn't collect testing data. Before disregarding our early OFDM implementation, we decided to also conduct a sea test, as we would suffer less reverberations and would emulate real-world operating conditions better.

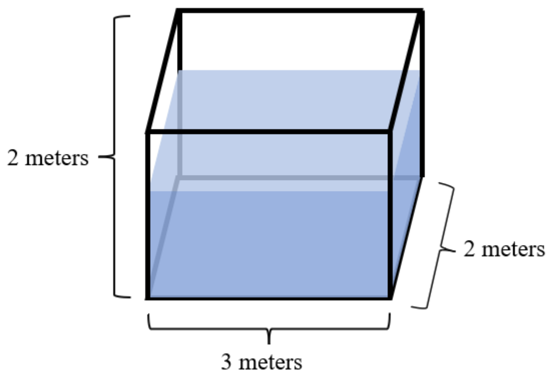


Figure 18: The approximate specifications of the pool

3.8.3 Sea test

Before a sea test could be conducted, a mobile power source was needed. The institute provided two 23.1V batteries, however the connection was not suitable for the modems, thus the appropriate connection had to be soldered onto the existing connections. In order to power the ethernet switch, a mobile AC power supply was borrowed from Vortex NTNU. Some rope was also borrowed from the lab to hoist the modems into the water. The test was conducted at a NTNU location in Nyhavna, Trondheim. The test was conducted from the floating pier, and the modems were lowered on each side of the pier. Figure 19 shows a diagram of the test.

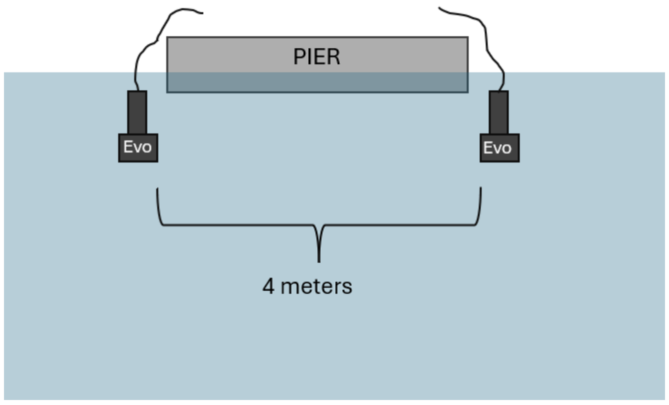


Figure 19: Diagram of the sea test

We were still using our early OFDM implementation, and once again the results were worse than in air. To make sure the channel conditions of the area actually were suitable for acoustic transmission, a test using the EvoLogics AMA software was ran. EvoLogics AMA was able to transmit and receive messages without problem. Thus our OFDM implementation was most likely the culprit. It was at this point decided to start working on a JANUS implementation, as we knew the previous group had had success using JANUS, and we wanted a robust protocol for our ra-NRC algorithm. The results from this test are not discussed in Results, as this test was purely for developmental reasons, and we didn't collect testing data.

3.8.4 River test

Towards the later days of development, a river test was conducted in Nidelva, Trondheim. This was after developing a successful JANUS implementation and a working ra-NRC algorithm. We originally wanted to conduct this test in the pool, but to our surprise the pool had been drained since the last time, and was not schedueled to be refilled for another few weeks. Although the testing conditions of the river were non-ideal, they still proved a successful proof of concept our our JANUS implementation and ra-NRC algorithm. We had hoped to gain access to one of the piers in the river in order to access deep water, but the piers were locked and we had to settle for doing a test along the river bank itself. Results will be discussed in Results.

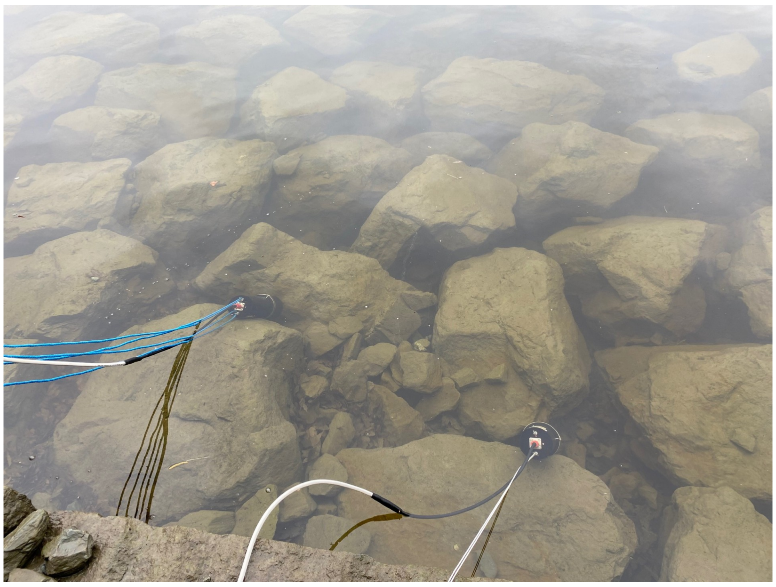


Figure 20: The river test. At the start of the test the water level was higher, but it decreased during the test.

4 Results

4.1 Modification of the cost function

This section shows how the changes made to the cost function affected the resulting parameters. The first three plots are done with the same network of three nodes. Figure 22 is before modification for reference, figure 23 has P as an additional optimization variable, figure 25 has the new optimization variable while also optimizing for successful bit rate per packet, figure 24 optimizes for successful bit rate per packet, but doesn't have P as an optimizable variable.

Running the simulator in debug mode was able to gives various results that don't appear in the plots: Manually changing P to its min and max values showed that SINR varies between 98.51 and 99.98, the constant noise value was $1.09 \cdot 10^{-9}$ W, and the constant absorption was $9.1 \cdot 10^{-12}$ dB/m. The network plots only represents the connection between nodes and has nothing to do with the range between them. For visibility purposes, N has its own plot despite being a part of *Node.xi*, and there are no labels to mark which node the optimization variables are part of as this was not seen important.

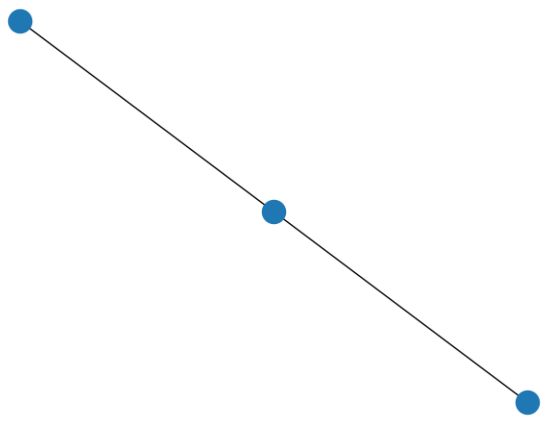


Figure 21: Randomly generated network used for all simulations using three nodes

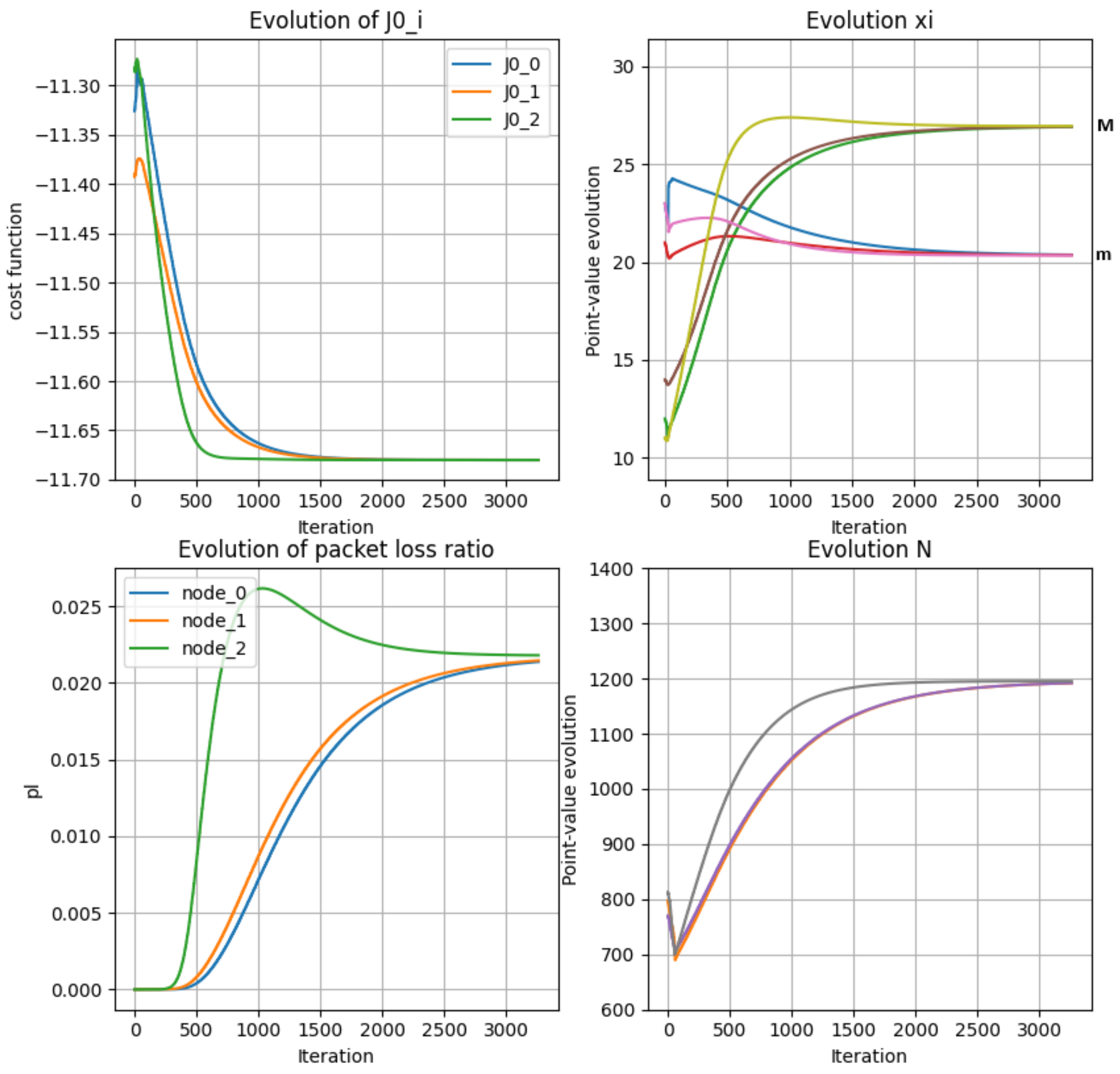


Figure 22: Using the old cost function.
Results: $J_0 \approx -11.68$ $p_l \approx 0.021$ $m \approx 20.4$ $N \approx 1200$ $M \approx 26.9$

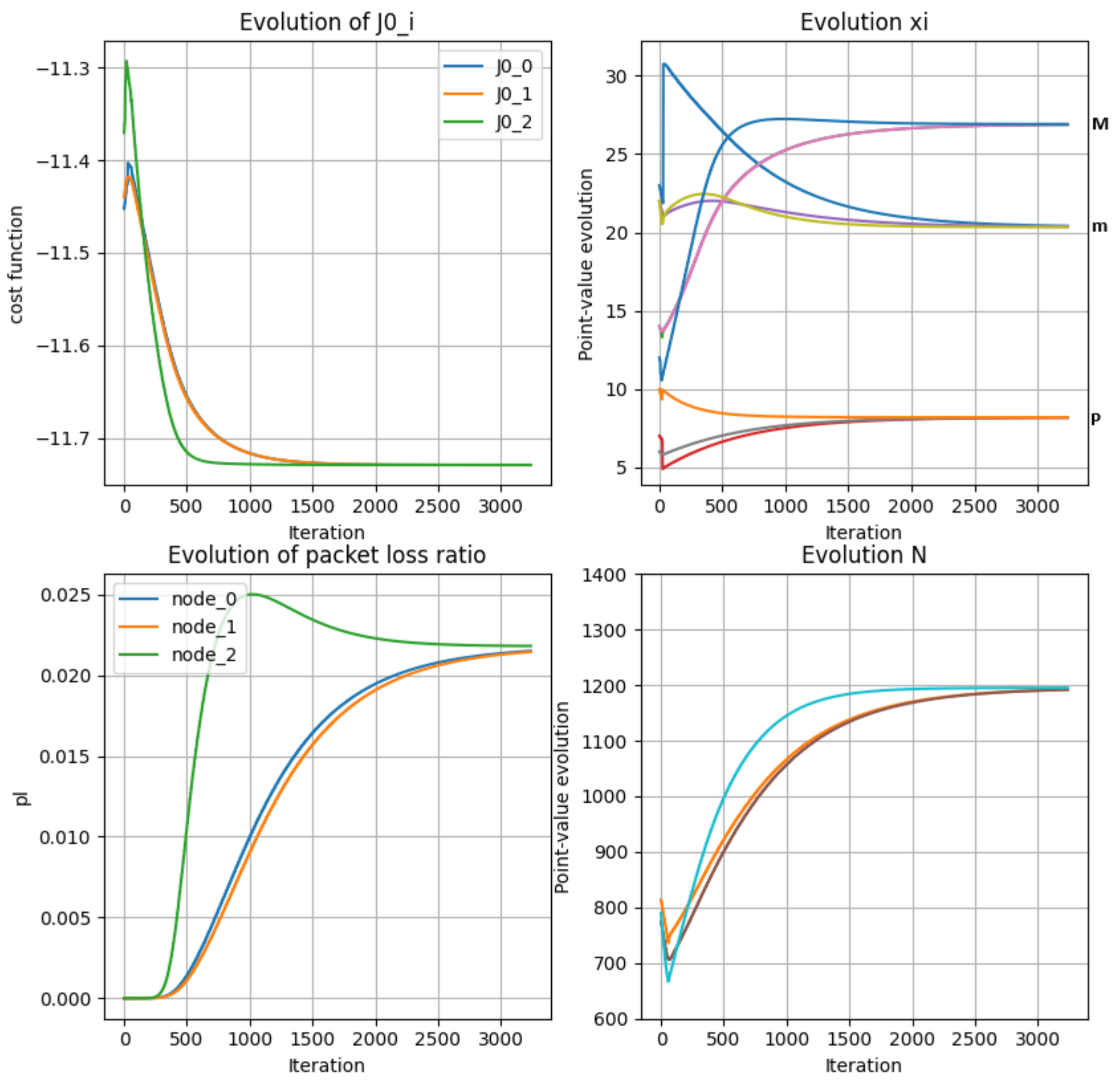


Figure 23: P is added as an additional optimization variable.
Results: $J_0 \approx -11.73$ $p_l \approx 0.021$ $m \approx 20.4$ $N \approx 1200$ $M \approx 26.9$ $P \approx 8.2$

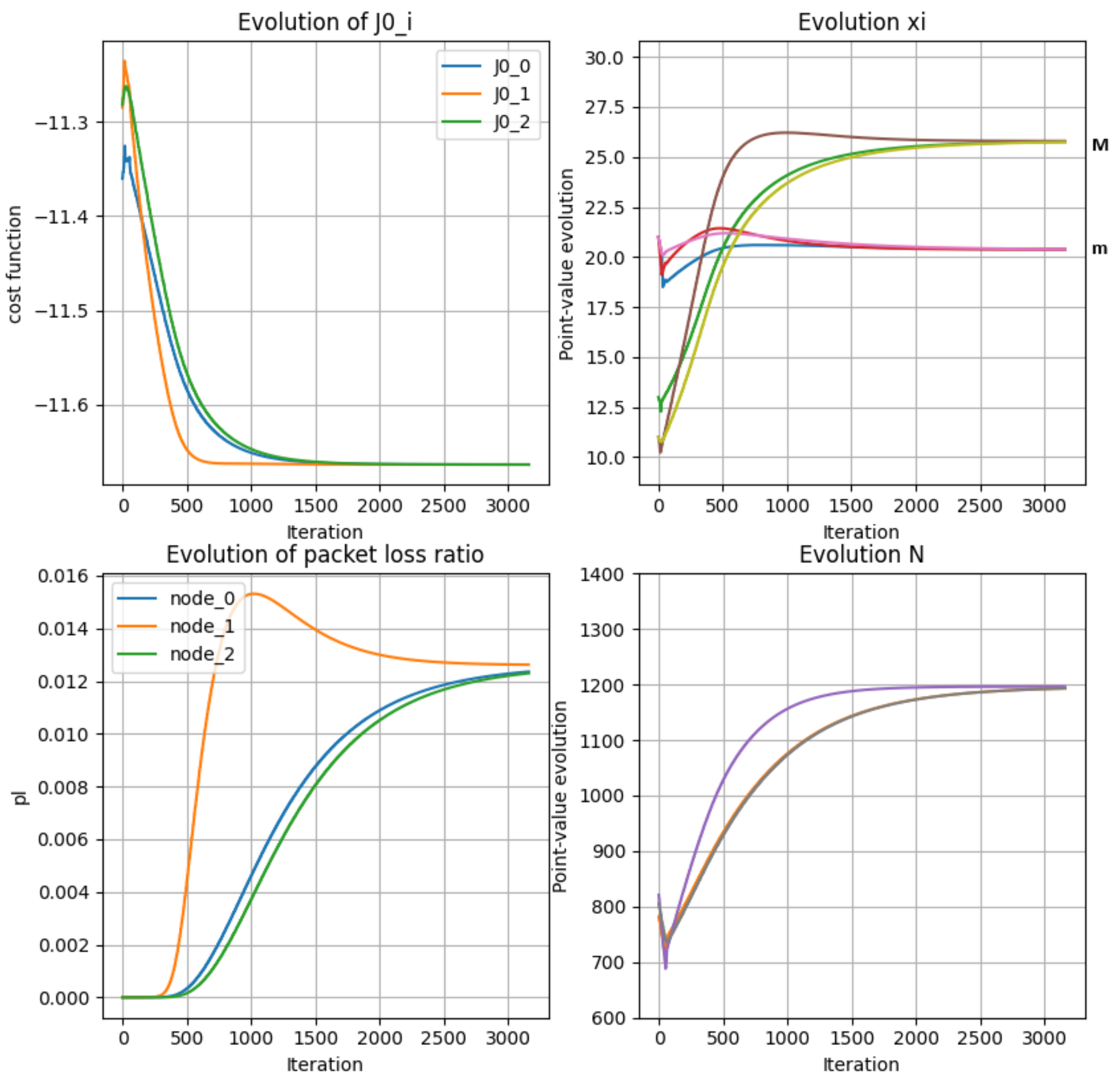


Figure 24: Optimizing for successful bit rate per packet, but without P as an optimizable variable. Results: $J_0 \approx -11.67$ $p_l \approx 0.012$ $m \approx 20.4$ $N \approx 1200, M \approx 25.8$

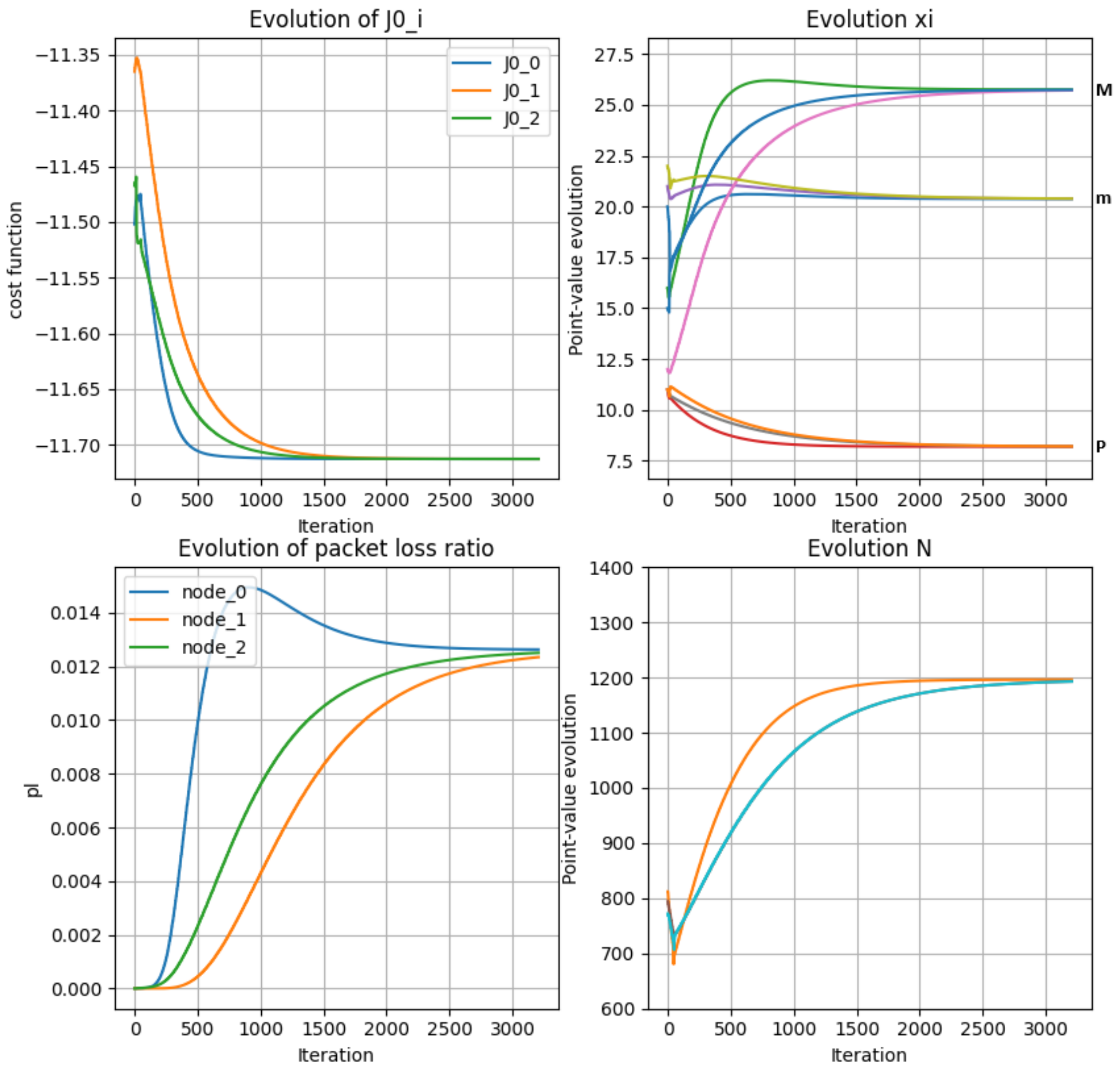


Figure 25: Optimizing for successful bit rate per packet with P added as an optimizable variable. Results: $J_0 \approx -11.71$ $p_l \approx 0.012$ $m \approx 20.4$ $N \approx 1200$ $M \approx 25.8$ $P \approx 8.2$

4.2 Results from air testing

As mentioned earlier, our initial air testing was unreliable. However after implementing JANUS, air testing became much more reliable, and we would get reproducible results. The JANUS encoding and decoding scripts seem to work without problem. Using *sdmsh* to transmit a JANUS packet can result in a packet success rate of up to 98 percent. Using *sdm* however, was more unreliable. Using *sdm* we were able to achieve a packet success rate of 80 percent during 200 transmissions. As the *sdm* simply is the Python conversion of *sdmsh*, it became unclear to us why they would produce different results. The reason for this difference in results is unclear, but by isolating different parts of our code it seems to be a problem with the *sdm* library itself.

4.2.1 Results from ra-NRC

Several air tests of the ra-NRC algorithm were ran. The successful packet rate of the tests would differ between 60 percent at its worst and 90 percent at its best, for iterations larger than 30. In order to test if the ra-NRC algorithm would converge we ran a test of 50 iterations using a simplified cost function which we would expect the function value to converge to 0.75. A plot showing the evolution of the function value for each iteration can be shown in Figure 26

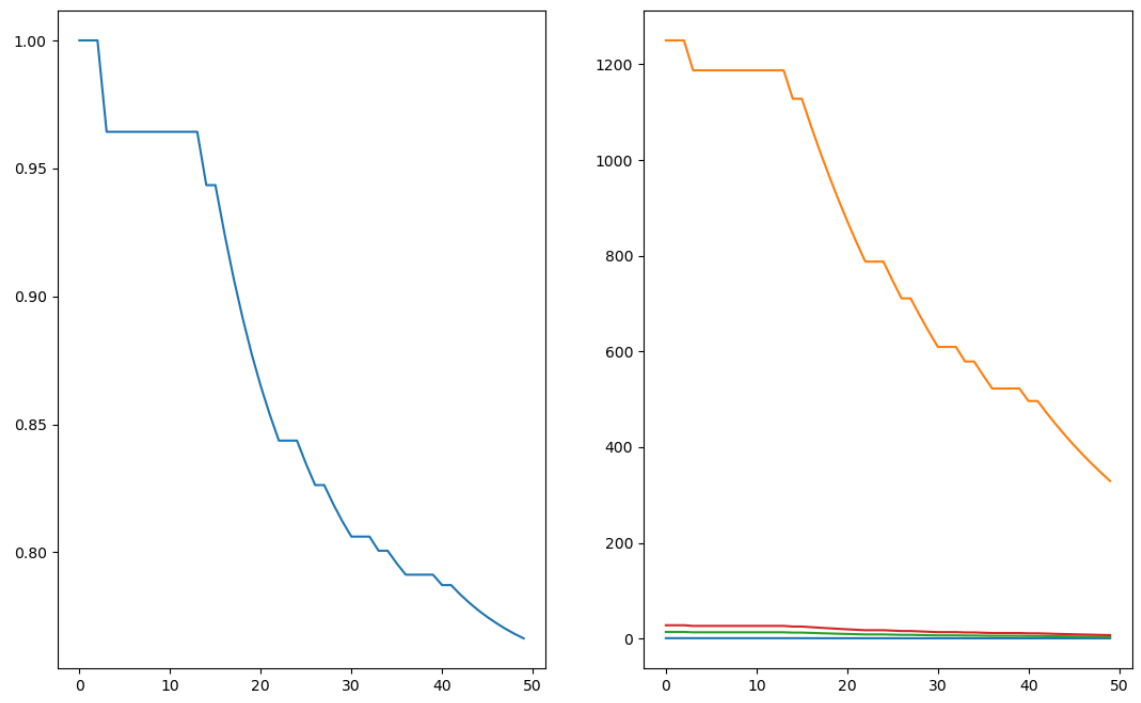


Figure 26: ra-NRC air test using simple cost function. The values are retrieved from a single modem.

The function value in the left plot can be seen starting to converge towards 0.75, and this was with a packet success rate of approximately 75 percent, highlighting that convergence isn't neccessarily dependent on perfect transmissions.

Tests were also run using the cost function we developed ourselves, not the simplified one. This test needed more iterations, as the convergence is usually reached after several thousand iterations. Simply increasing step size would add numerical instability to our update estimation, so we conducted a test using 200 iterations, a test of several thousands iterations were not realistic due to the time it took to transmit and receive. The results can be seen in Figure 27.

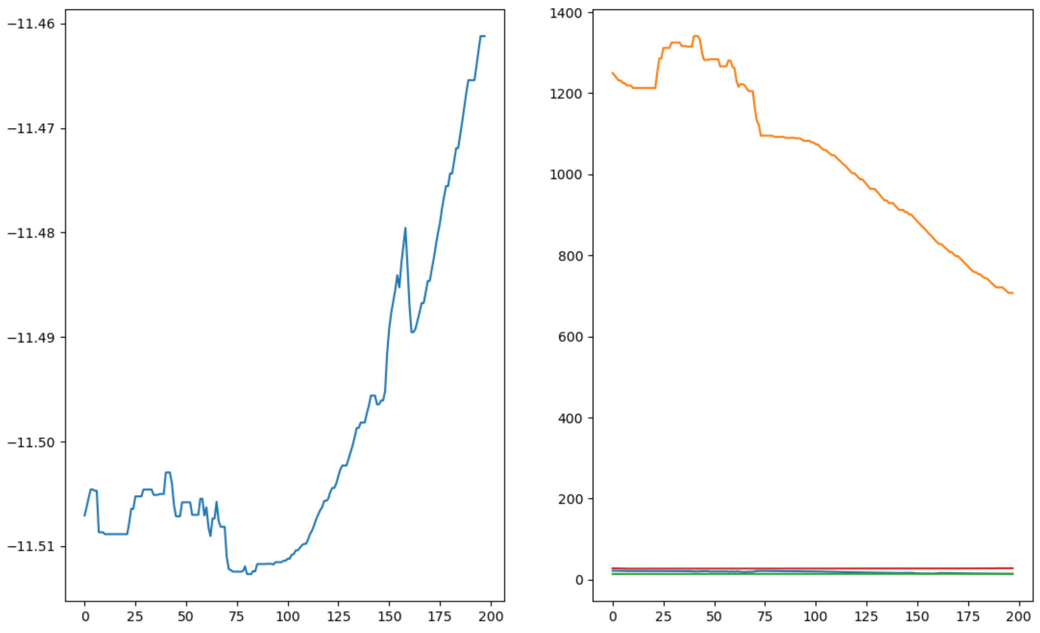


Figure 27: ra-NRC air test using our own cost function. The values are retrieved from a single modem.

The results of this test are less satisfactory, as the function value doesn't seem to be converging but instead diverging. However, as can be shown in the simulated plots, Figure 25, 24 and 23. some numerical instabiltiy is to expected for the first 500 iterations or so. Therefore our 200 iterations aren't necessarily enough to escape the initial instability of the algorithm. It is also worth noting that the graph doesn't start at zero. The steep increase only represents a 0.05 increase in function value.

4.3 Results from the river test

During the river test we ran three different tests. Due to the nature of the acoustic channel in the river, we first had to experiment with which pre-amplifier gain and which source level to use. We landed on a pre-amplifier gain of 40 and a source level of 3 (minimum). As we had limited time, we only ran 15 iterations of ra-NRC, and the results from the final test can be shown in Figure 28.

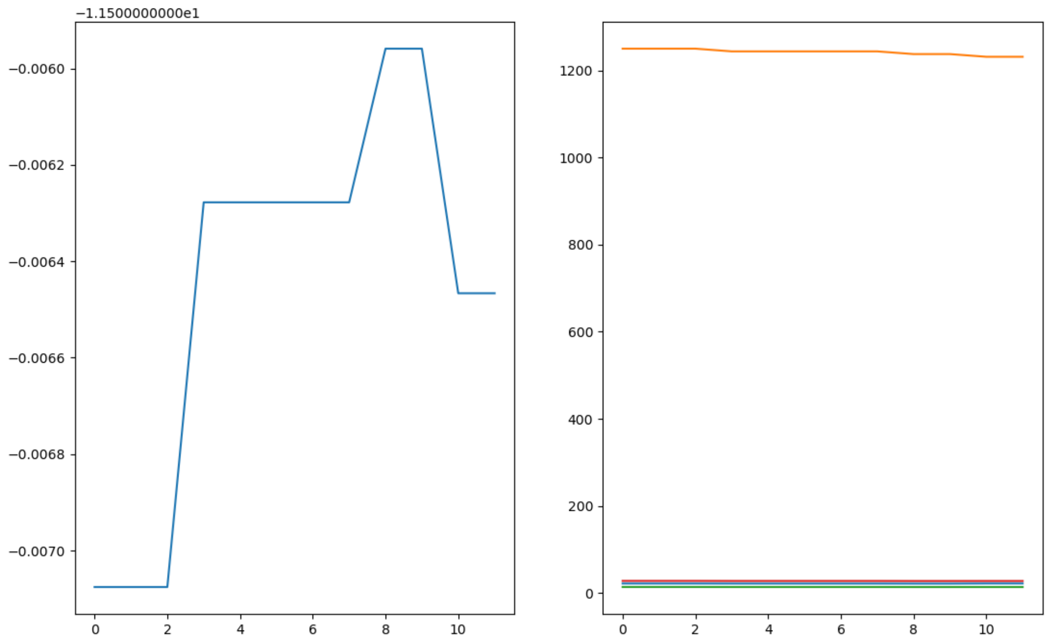


Figure 28: ra-NRC river test using our own cost function. The values are retrieved from a single modem.

We only achieved a packet success rate of 66 percent. The evolution of the function value does not seem to converge, which is to be expected, as we only ran 15 iterations when several thousand are needed. The test worked more as a proof of concept that our system had a potential to be used in the field. Even though the packet success rate was 66 percent, the bit success rate was closer to 88 percent. This discrepancy in packet success rate and bit success rate makes sense, as a single bit error in decoded cargo will result in a complete packet fault. The testing conditions of the river were non-ideal, the modems were closely spaced together and surrounded by rocks. The modems didn't have perfect line of sight to each other either.

4.4 The system

The final goal of the thesis was to create an underwater communication system that enables users to implement distributed optimization algorithms as they wish, and in this way contribute to the development of multi-agent autonomous solutions in the subsea environment.

The results we got in our work are thus not only measured in numbers and graphs, but also as how high we have succeeding in pushing the technology readiness levels (TRL) of such a system.

Striving for achieving the highest possible TRL has pushed us thinking about how the system itself should be structured. The resulting structure, that we envision as a result, is that the system seems best to be separated into two parts, the ra-NRC part and the JANUS part, so to enable concurrent and modular development of the constituting parts.

4.4.1 ra-NRC_main.py

More in details, the optimization routines have been collected in the code in the class above, that implements a *Node* class that provides a method of using the ra-NRC algorithm alongside JANUS. The user may select any cost function, as long as the transmitted variables to not exceed an array of four or a matrix of 4x4. The user can also select number of iterations, step size, minimum accepted convergence and other parameters. Through testing, the best successful packet rate achieved was 95 percent over 50 iterations. The system requires the user to initialize the modems using terminal commands first, but once the script has started the system is autonomous and the modems communicate between each other until the maximum number of iterations has been reached.

4.4.2 JANUS implementation

Our JANUS implementation is very similar to the Tool Kit from which it was derived from. Our version differs by working continuously from a single terminal command, whereas the original required a terminal command for each packet to be generated. Our version also does dual-packet generation and decoding. The JANUS implementation is not limited to work with the ra-NRC. On the tx side, it simply reads from *shared_string.txt* and produces packages from its contents, meaning any program can create packages. On the receiving side it simply reads packages and prints its contents into *cargo_data1.txt* and *cargo_data2.txt*. Our JANUS implementation is therefore flexible to work with other programs than ra-NRC.

4.4.3 OFDM implementation

The final implementation of the OFDM algorithm is a software stack that can be used to transmit data using the parameters from Table 2. It allows for transfer of both numerical arrays and image data.

As it is written to be user-friendly in object-oriented programming using python, the system is designed to be maintainable and scalable. It allows for easy modification of code functionality based on the specific requirements of the desired communication and optimization scheme in the future. The implementation however, does not currently interface with the consensus algorithm, which was a goal for the system. The reasons are discussed in Results.

5 Discussion

5.1 Cost function

The results show that both before and after adding P as an additional variable, there are not many changes; only M has become slightly lower:

$$m \approx 20 \quad M \approx 27 \rightarrow 26 \quad N \approx 1200 \quad (P \approx 28 \text{ for figure 23})$$

In both cases the packet loss ratio is about 0.02. One might notice that the cost function seems to have a lower value in figure 23 and 25, but this is due to the introduction of two additional barrier functions (one each for min and max P), which are rather badly approximated with a t of only 10. Because all the plots uses a number of badly approximated barrier functions, each plot appears to give a more optimistic value than it actually is if the barrier function were more accurate.

Comparing the result after adding P, the change is basically nonexistent. This is to be expected as the amount SINR doesn't vary all that much when changing P. The same is seen when comparing figure 24 and 25. Comparing the old cost function to the one optimizing for successful bit rate per packet, there are a few differences of note. First is that the final cost is a tiny bit smaller after the change. Second is that M is about one lower, causing the packet loss ratio to drop by almost half. This makes sense as the cost function now scales with one minus the packet loss ratio, making it constantly want to keep the packet loss ratio down, instead of just having it as a restriction.

The following comparison is based on the old cost function and the one optimized for successful bit rate per packet without P so that the barrier function gives an equal amount of "fault" to both values. The actual calculation will also use more accurate values than the ones displayed. To see how the bit rate is affected by this change, both values are converted to their positive non-logarithmic value, then the modified function is scaled such that it gives pure bit rate. The bit rate gained will be the difference between the two. To see how the successful bit rate changes, the old cost function has to be scaled instead:

$$\begin{aligned} &e^{11.67}/(1-0.012) - e^{11.68} \approx -496 \quad \text{difference in bit rate} \\ &e^{11.67} - e^{11.68} \cdot (1-0.021) \approx 576 \quad \text{difference in successful bit rate per packet} \end{aligned} \tag{45}$$

This means that while the overall bit rate is lower, the amount that can be used (successful bits) is higher. Again, these values are inaccurate because of the barrier function, so the loss and gain are not as high as shown.

The goal of modifying the cost function was to increase the bit rate that actually can be used by the receiver and improving estimation of SINR by optimizing P and including noise. As the results show, the goal of modifying the cost function has at least been achieved in simulations. The effects of improving SINR has been difficult to interpret as there didn't seem to be many changes.

If one were to want more accurate results, which would require a better t, the interior point method could be utilized as mentioned in section 2.1. But because using IPM requires some extra work with finding the best number iterations per cycle, and how much to increase t with per cycle. Because the accuracy it provides in exchange isn't of much importance for the results this project was after, it was deprioritized. The simulation varied somewhat where the values converged each time it was simulated, so the result is more a guideline of what one can expect rather than an exact measure.

There are several things that should also have been tested, but weren't due to a lack of time. Adding environmental factors as node specific instead of constants in the cost function itself could have given a look at how the system adapts when the environment is different for each node. Experimenting with different constant values for SINR when P wasn't being optimized could have given a clearer image of the effects that changing P brings; having a constant SINR value of 100, which is pretty similar to what it is when it is being optimized, makes it hard to see any changes.

This is a theoretical result that assumes the parameters can be varied continuously. However as mentioned in section 2.6.2, some parameters can only be certain values; for example the modulation order can only be a power of two. The resulting parameters are therefore not necessarily possible in the real world, meaning a rounding has to be done, which can make the result vastly different from the simulations. A better solution in finding the best OFDM parameters would require some implementation such that it is accounted for that the variables can only be certain values.

The cost function itself has room for improvement in several areas. The noise model is assumed constant and based on assumed background noise. An improvement could be incorporating several sources for noise, either by modelling or by measuring the noise profile, or a combination of both. All the assumed values could also be estimated more accurately. Another improvement that relates both to the simulations and to the system as a whole is the required iterations. As it stands the system requires several thousand iterations for each node, which in simulations is manageable, but infeasible in practice. A possible solution can be experimenting with the step size and number of iterations, though this can lead to instabilities and inaccuracies.

5.1.1 Values jumping "out of bounds"

One problem that sometimes occurred with simulation was that the cost function would become "nan" (not a number), which would render both the plots and results useless. This was tracked down to one or multiple of the log barrier functions trying to take the logarithm of a negative number, which means that the variable that is being restricted has a value outside the allowed interval. By the definition of the barrier function, the "penalty" should be infinitely high, but artificially forcing a high punishment in the case of an illegal value turned out to just cause more instabilities and break the convergence.

It was never consistent which of the functions this happened with, and while it rarely happened with small networks, larger networks of ten nodes was almost impossible to simulate because of this. This problem occurred with the old cost function, but became much more frequent after doing either (or both) of the modifications. The cause of this is not known, but intuition tells that the added complexity makes for a more unstable convergence, especially when either modification adds an additional exponential term to calculate (44 and 36), which causes the function to act more aggressively.

5.2 High packet error in ra-NRC

5.2.1 sdmsh

When running air tests of ra-NRC one could expect packet success rates of between 60 and 90 percent during a session. A recurring problem was that for each dual-packet transmission, only the second packet was received and decoded properly. The first packet would not trigger any JANUS detection, while the second packet would get decoded perfectly. By isolating the problem it seems to point to an issue with how *sdm* transmits packages. Running the JANUS servers with *sdmsh* instead would produce almost perfect results. By fixing this issue one could expect at least a 30 percent increase in successful packet rate.

Initially the group thought using Python for interfacing with the modems would be the simplest and best way to move forward. Partly due to a lot of experience in Python, and lack thereof in "C". In retrospect, the simplest path was not probably the best. The simplicity of the Python libraries proved more to be a hindrance than an asset. Also, having to work with "C" anyways for the JANUS implementation provided a lot of experience in "C". If the group could start from the beginning, not using *sdm* and instead interfacing with the underlying "C" libraries of *sdmsh* directly would probably be a better way to go.

5.2.2 "DROP" error

Also highlighted by the previous group, "DROP" errors became common during longer sessions. We discovered an almost complete fix of this problem by ensuring a *send_stop()* between all commands. Though encountered rarely, *sdm* will somtimes report "DROP" errors and freeze mid transmission. Although our watchdog timer will reset the modem, a packet will still get lost and time will get lost. Figuring out the root cause of the "DROP" error can lead to extra time saved and a better packet success rate.

5.2.3 JANUS cargo

Per the Tool Kit manual, the only accepted way of formatting JANUS cargo seemed to be as a string. Being forced to format the cargo as a string led to some difficulties in decoding the strings. Converting the string contents into variables would sometimes produce unseen formatting errors. Some packet's cargos would appear perfect, without any bit error, but still get rejected by the expected formatting. A significant amount of time was spent on finding a way to decode the strings into float arrays, but in the end our method still proved to be unsatisfactory, leading to an unnecessary amount of packets getting discarded.

5.3 Time spent per iteration

The modems spend significant time per iteration of ra-NRC. One of the reasons is the lack of synchronization with the JANUS servers. Right now *ra-NRC_main.py* is forced to simply wait for X amounts of seconds, with synchronization between ra-NRC and JANUS this timeout can be reduced.

A transmission itself also takes a few seconds. This is partly due to the small waiting window between two packets. The current waiting window is 1 seconds and can probably be lowered. Reducing the sampling rate of the JANUS packets will also reduce the amount of samples per packet, resulting in a shorter transmission time.

The reception is hard coded to listen for X amount of samples. This X is set to have a safety margin so no samples get lost. This safety margin does however produce some extra time spent listening. Instead of listening for X amount of samples, a method of realizing when the reception is done can be developed. This would shorten down reception times. Doing these changes can probably double the amount of iterations which can be done in a given time span. As of right now, the modems function at a rate of approximately 6 iterations per minute, meaning for our largest test of 200 iterations - our ra-NRC algorithm ran continuously for 33 minutes.

5.4 Modems requiring hard resets

Sometimes, the modem would seemingly freeze and become unresponsive, even to *stop()* commands. The only solution for this seemed to be to pull the plug. The origin of this error is unidentified. A great annoyance of this error is the fact that you have to reinitialize the modems using the terminal after a power on, as this isn't automated.

5.5 Non-ideal tests

Although hundreds of air tests were performed, the group only performed three field tests, if you count the pool test. Initially the group wished to perform more sophisticated field tests, but the development process proved itself lengthier and more difficult than expected. One of the tests the group considered important was what would be a final test in the pool, where the group would run hundreds of iterations of ra-NRC. However, when the group decided to perform these tests, the pool had surprisingly been drained. Thus the group were forced to perform a test outside instead. Due to logistical reasons and time constraints a river test seemed to be the best option. The group was optimistic in the access to piers in the river, but upon arrival noticed the piers were locked. Having to do the tests along the river banks was non-ideal, as the group knew the modems are sensitive to weird geometries like river rocks and reverberations due to tight spaces. The ideal case would be to perform an ocean test were the modems are able to be lowered far below the surface and far apart, free from aggressive reverberations, and more close to actual operating conditions.

5.6 Development process

As the group had little to no prior experience in the field of acoustics, a lot had to be learned and discovered on the way. Initially the group thought the development process would be easier as a lot of implementations already existed. However, compared to other programming disciplines, programming of acoustic modems seemed to be a niche and not a lot of resources were available online. For the existing implementations the group found, they were either specialized or contained little to no documentation. The development process thus turned out harder than expected. The lack of documentation lead to sort of development process where one would have to "hack" already existing implementations, figure out how they work by trial and error, and extract the relevant functions.

5.7 Lack of OFDM integration

Because of development time for the OFDM part of system, the project was severely constrained on time in the end. This lead to the ra-NRC consensus system not being integrated with the

OFDM implementation, which was one of the goals of our project. The same time constraints also limited the amount of testing we were able to do with our OFDM implementation, thus leading to the lack of test results in Results. For future work, the ra-NRC system should be made to pass its parameters to the OFDM implementation after reaching consensus. We hypothesize that using a ROS (Robot Operating System) for this tie-together is an optimal strategy.

5.8 UN's goals for sustainable development

Regarding goal 9, this project can be said to have had a positive effect. The project has exercised innovation regarding improving performance and efficiency in the field of underwater communication and multi-agent autonomous systems. This in turn contributes to creating future jobs and industrialization opportunities, as the technology readiness level has been increased. The project can also be said to have had a positive effect regarding goal 14. By improving the automation of underwater systems, it becomes easier and more sustainable to manage and control underwater technology and installations. The need for manned surveillance or use of underwater drones decreases, which frees up resources and minimizes the need for doing operations that may be disruptive to the marine life.

6 Conclusion

This thesis has demonstrated the practical implementation of an underwater communication system utilizing distributed optimization algorithms. By leveraging the JANUS standard and integrating it with a sophisticated ra-NRC optimization algorithm, the project achieved a significant step forward in the development of multi-agent autonomous communication systems for subsea environments.

Separating the system into the ra-NRC optimization part and the communication part has allowed for concurrent and modular development, enabling future enhancements and scalability. The system's performance, as indicated by the air and water tests show the potential for functional distributed algorithms, if one were to invest more time into fixing the various problems that have been discussed.

Despite some challenges, such as inconsistencies between Python and shell script implementations and initial numerical instability in the optimization process, the results are promising. The optimization for successful bit rate per packet, although requiring further fine-tuning, has demonstrated an increase in the amount of data successfully received, which is crucial for reliable underwater communication.

In conclusion, this project not only advances the technology readiness level of underwater communication systems but also provides a flexible and robust framework for future research and development. The successful integration of distributed optimization algorithms with underwater acoustic modems sets a solid foundation for more sophisticated and autonomous subsea communication solutions.

Bibliography

Armstrong, Jean et al. (Nov. 2006). 'Performance of Asymmetrically Clipped Optical OFDM in AWGN for an Intensity Modulated Direct Detection System.' In: DOI: 10.1109/GLOCOM.2006.571.

Bof, Nicoletta et al. (2019). 'Multiagent Newton–Raphson Optimization Over Lossy Networks'. In: *IEEE Transactions on Automatic Control* 64.7, pp. 2983–2990. DOI: 10.1109/TAC.2018.2874748.

Boyd, S. and L. Vandenberghe (2004). *Convex Optimization.* Cambridge University Press. ISBN: 9781107394001. URL: https://books.google.no/books?id=IUZdAAAAQBAJ.

Chen, Si and Alexander M. Wyglinski (2010). 'Chapter 3 - Digital communication fundamentals for cognitive radio'. In: *Cognitive Radio Communications and Networks.* Ed. by Alexander M. Wyglinski, Maziar Nekovee and Y. Thomas Hou. Oxford: Academic Press, pp. 41–83. ISBN: 978-0-12-374715-0. DOI: https://doi.org/10.1016/B978-0-12-374715-0.00003-4. URL: https://www.sciencedirect.com/science/article/pii/B9780123747150000034.

Cho, Jin-Ho and Sang Woo (Oct. 2011). 'Communication Strategies for Various Types of Swallowable Telemetry Capsules'. In: ISBN: 978-953-307-415-3. DOI: 10.5772/23924.

Coates, Rodney F. W. (1990). *Underwater Acoustic Systems.* Red Globe Press London. DOI: 10.1007/978-1-349-20508-0.

'Distributed Optimization Based Adaptive Underwater Communication Schemes' (2023). MA thesis. Norwegian University of Science and Technology.

Dosits (n.d.[a]). *Introduction to Decibels.* URL: https://dosits.org/science/advanced-topics/introduction-to-decibels/ (visited on 1st Feb. 2024).

— (n.d.[b]). *Introduction to Decibels.* URL: https://dosits.org/science/advanced-topics/source-level/ (visited on 1st Feb. 2024).

EvoLogics (2018). *S2C R 18/34 USBL Underwater Acoustic Modem.* URL: https://evologics.com/product/s2c-r-18-34-usbl-2s (visited on 5th Mar. 2024).

— (2021). *sdmsh: Commands and Parameters.* URL: https://github.com/EvoLogics/sdmsh/wiki/sdmsh-%3A-Commands-and-Parameters (visited on 5th Mar. 2024).

Firdaus, Muhammad and Yoedy Moegiharto (2022). *Performance of OFDM System against Different Cyclic Prefix Lengths on Multipath Fading Channels.* arXiv: 2207.13045 [cs.NI].

Fisher, F. H. and V. P. Simmons (Sept. 1977). 'Sound absorption in sea water'. In: *The Journal of the Acoustical Society of America* 62.3, pp. 558–564. ISSN: 0001-4966. DOI: 10.1121/1.381574. eprint: https://pubs.aip.org/asa/jasa/article-pdf/62/3/558/12209639/558_1_online.pdf. URL: https://doi.org/10.1121/1.381574.

Goldsmith, A.J. (2005). *Wireless Communications.* Cambridge University Press.

Goldsmith, A.J. and Soon-Ghee Chua (1997). 'Variable-rate variable-power MQAM for fading channels'. In: *IEEE Transactions on Communications* 45.10, pp. 1218–1230. DOI: 10.1109/26.634685.

Iadarola, F. (2022). 'Multi-agent algorithms for adaptation of underwater acoustic communication parameters'. MA thesis. University of Bologna.

Kuisma, Mikael Q. (2023). 'I/Q Data for Dummies'. In: URL: http://whiteboard.ping.se/SDR/IQ (visited on 20th May 2024).

Li, Baosheng et al. (2009). 'MIMO-OFDM for High-Rate Underwater Acoustic Communications'. In: *IEEE Journal of Oceanic Engineering* 34.4, pp. 634–644. DOI: 10.1109/JOE.2009.2032005.

NATO (2024). 'ANEP-87'. In: *NATO Standardization Office.*

Pérez-Neira, Ana I and Marc Realp Campalans (2009). 'Orthogonal frequency division multiplexing'. In: *Cross-Layer Resource Allocation in Wireless Communications.* Elsevier, pp. 151–162. DOI: 10.1016/b978-0-12-374141-7.00008-7. URL: http://dx.doi.org/10.1016/B978-0-12-374141-7.00008-7.

Prasad, Ramjee (2004). Artech House, p. 272. ISBN: 1580537995, 9781580537995.

Radosevic, Andreja et al. (2014). 'Adaptive OFDM Modulation for Underwater Acoustic Communications: Design Considerations and Experimental Results'. In: *IEEE Journal of Oceanic Engineering* 39.2, pp. 357–370. DOI: 10.1109/JOE.2013.2253212.

Schirripa Spagnolo, Giuseppe, Lorenzo Cozzella and Fabio Leccese (Apr. 2020). 'Underwater Optical Wireless Communications: Overview'. In: *Sensors* 20.8, p. 2261. ISSN: 1424-8220. DOI: 10.3390/s20082261. URL: http://dx.doi.org/10.3390/s20082261.

Sehgal, Anuj, Iyad Tumar and Jürgen Schönwälder (June 2009). 'Variability of available capacity due to the effects of depth and temperature in the underwater acoustic communication channel'. In: pp. 1–6. DOI: 10.1109/OCEANSE.2009.5278268.

Stojanovic, Milica (2006). 'Low Complexity OFDM Detector for Underwater Acoustic Channels'. In: *OCEANS 2006*, pp. 1–6. DOI: 10.1109/OCEANS.2006.307057.

UN (2023a). *Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation.* URL: https://sdgs.un.org/goals/goal9 (visited on 10th May 2024).

— (2023b). *Conserve and sustainably use the oceans, seas and marine resources for sustainable development.* URL: https://sdgs.un.org/goals/goal14 (visited on 10th May 2024).

Varagnolo, Damiano et al. (2016). 'Newton-Raphson Consensus for Distributed Convex Optimization'. In: *IEEE Transactions on Automatic Control* 61.4, pp. 994–1009. DOI: 10.1109/TAC.2015.2449811.

Wengle, Emil et al. (2024). 'Experimental assessment of a JANUS-based consensus protocol'. In: *Computer Networks* 244, p. 110345. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2024.110345. URL: https://www.sciencedirect.com/science/article/pii/S1389128624001774.

Wenz, Gordon M. (Dec. 1962). 'Acoustic Ambient Noise in the Ocean: Spectra and Sources'. In: *The Journal of the Acoustical Society of America* 34.12, pp. 1936–1956. ISSN: 0001-4966. DOI: 10.1121/1.1909155. eprint: https://pubs.aip.org/asa/jasa/article-pdf/34/12/1936/18746565/1936_1_online.pdf. URL: https://doi.org/10.1121/1.1909155.

Appendix

A poster

Distributed algorithms via acoustic marine communication gateways

Background

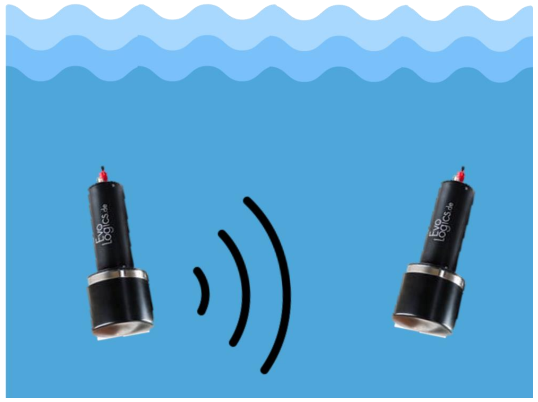
Underwater communication is a growing field of study. Due to the absorption properties of water, electromagnetic waves such as radio waves are non-ideal for long-distance communication underwater. Sound, however, travels faster underwater than in air and does not get absorbed by water as easily. Therefore, communicating between nodes using acoustic modems is a viable method for underwater communication.

Results

We have expanded an existing cost function to account for more OFDM parameters. Our system enables two nodes to communicate with each other to optimize using this cost function. By conducting both air and water tests, we have demonstrated that our system can reach consensus and has the potential to be used in real-world applications.

Our thesis

Our thesis explores the topic of distributed optimization using acoustic modems for underwater nodes. By utilizing a cost function for the bit rate of an underwater OFDM transmission and enabling nodes to optimize using the robust asynchronous Newton-Raphson Consensus, two or more nodes can reach a consensus on the optimal OFDM parameters for achieving the best transmission bit rate.



Bachelor thesis: Ingar Eik Høivik, Thomas Aleksander Jonsson and Benjaminas Visockis

NTNU

May 2024