Faculdade de Engenharia da Universidade do Porto

U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Integration of optical communications in an underwater docking station

Helder José Baldaia Esteves

FINAL VERSION

Dissertation formulated in the scope of
Master's in Electrical and Computer Engineering
Automation major

Supervisor: Nuno Alexandre Lopes Moreira da Cruz
Co-supervisor: José Carlos dos Santos Alves

February of 2018

Abstract

Underwater optical communication is a recent emerging technology that allows autonomous underwater vehicles to transfer information at close-range efficiently. It offers a higher data-transfer rate and a lower latency than acoustic means. For this reason, this technology shows potential over short-range communication, improving cases such as the underwater docking maneuvers by adding a feedback system. However, the harshness of the underwater environment poses a new problem to the reliability of this mean. This dissertation introduces a communication protocol based on an error-correction control mechanism. The proposed solution implements a hybrid automatic repeat request type II with incremental redundancy. Reed-Solomon was used as part of the forward error-correction. Comparisons between common usage of fixed code rates and the proposed solution feature an increase of up to 45% in efficiency for the latter, while maintaining the same decoding capability as the lowest fixed code rate. Results also report that the error control system can handle localization errors from the AUV effectively, showing that the system is able to correct locations related to a difference of 2 encoding bytes effectively.

Contents

Figure list

Table list

x

Symbols and Abbreviations

Abbreviation list

ACK	Acknowledge
APD	Avalanche photodiode
ARQ	Automatic repeat request
AUV	Autonomous underwater vehicle
AWGN	Additive white Gaussian noise
BCH	Bose–Chaudhuri–Hocquenghem
BER	Bit error rate
BSC	Binary symmetric channel
COBS	Consistent overhead byte stuffing
FEC	Forward error correction
HDLC	High-level data link control
IDE	Integrated development environment
IR	Incremental redundancy
LDPC	Low-density parity-check
LED	Light emitting diode
NAK	Not acknowledge
OOK	On-off keying
PC	Personal computer
PD	Photodiode
PPP	Point-to-point protocol
ROV	Remotely operated vehicle
RPi2	Raspberry Pi 2
RS	Reed-Solomon
Rx	Receiver
SBL	Short baseline
SER	Symbol error rate
SNR	Signal-to-noise ratio
Tx	Transmitter
UART	Universal asynchronous receiver-transmitter
UWOC	Underwater optical communication

Symbol list

μ	Mean of a normal distribution
σ	Variance of a normal distribution
H	Efficiency

Chapter 1 - Introduction

It is known that the surface of the earth can be monitored easily at any time thanks to satellite technology. This is not the case for the underwater environment, as light has difficulty penetrating the depths of seas and oceans. As a result, technology specifically tailored to monitor this environment has been developed. Autonomous underwater vehicles (AUV) and remotely operated vehicles (ROV) are forms of unmanned underwater drones which can perform tasks with or without human intervention. This technology has shown to be at a level where it is possible to autonomously perform various underwater missions such as controlling the state of the surrounding environment. Among these two types of drones, AUVs are object of increasing development since ROVs need cables to operate, which limits their ability in scenarios that, for example, have many obstacles in the surroundings. Notwithstanding, although AUVs have more potential, not only the battery life is still a very restrictive factor for this technology, but also communication faces a new challenge without cabled connections. This implies that they need to surface to be able to recharge, taking more time to complete an operation and driving the costs higher. An underwater docking station, where the AUV can dock to recharge, exchange information and receive new orders is an emerging concept that has established its potential during the last two decades [1]. This development reduces the cost and avoids handling of the surfacing of the AUV which proves dangerous not only in turbulent waters, but also in situations where there is high naval traffic. Nevertheless, the challenge of maintaining efficient and reliable underwater communications remains for the AUV technology. The most widespread solution for this problem comprises of acoustic methods [2]. Although this method handles long distance communication reliably, it is not used for short-ranges due to high latency and low data rate. Since short-range communication is not made acoustically, close-range positioning systems are also limited. For this reason, the docking of AUVs in underwater stations requires different approaches that do not rely solely on acoustic positioning mechanisms [3]. Still, although these strategies prove effective under normal circumstances, during times where changing currents or even unknown events happen, AUVs would benefit from having a feedback mechanism or a communication system while docking [4,5].

A newer technology, optical communication, became a solution for fast short-range communication [6]. It is based on the simple idea of using light to transfer information from one end to the

other. This can be used not only to transfer information at high data rates, but also to aid the docking sequence substantially in unpredictable situations. Compared to acoustic, optical communication allows for a much greater data rate, making this option very viable in underwater environments in the near-future. Nonetheless, while it presents great advancements in the AUV technology, it is highly dependent on its implementation process. For this motive, they require careful planning to avoid losing or transferring incorrect information. Research involving robust communication protocols have been made for cabled and airborne networks, however, the uniqueness of the underwater environment poses a new set of problems [7]. This environment calls for efficient and reliable communication protocols, with well-thought designs capable of handling different limitations such as:

- Variable scattering of light photons and water absorption, lowering the received power.
- Environmental noise, ranging from high signal-to-noise ratio (SNR) in the deep ocean to low SNR in the coastal waters.
- Different channel behaviors, with shadow zones completely impairing the communication, to simply bit to byte losses due to SNR.
- Battery-reliant applications, requiring low power designs to avoid unnecessary battery depletion.

As a means to surpass these obstacles with utmost efficiency, this dissertation proposes a robust communication protocol using optical means.

1.1 Objective

The main purpose of this dissertation is to develop a robust optical communication protocol between an underwater station and an AUV. Throughout the project, it is imperative that tests are conducted and performed in a wide variety of scenarios.

To reach this goal, a list of sub-goals is considered:

- System analysis, by understanding how and in which circumstances the communication should be available.
- Study and improve the communication link by adding adequate algorithms and techniques.
- Design of the communication protocol at transport layer that suits most optical devices.

Chapter 2 - Literature review

This chapter presents an overview of the major scientific advances for underwater docking strategies and error-correction coding.

2.1 Underwater docking strategies

The clear majority of AUVs existent in the present are of cruising type, which move predominantly in the horizontal axis and require a minimum forward velocity to maintain controllability. For this motive, most underwater stations are cone or cylinder-shaped, serving itself as a guide for this type of AUVs to dock [8,9]. Hovering type AUV's may move both horizontally and vertically, however, being a recent technology, docking strategies are still being developed.

2.1.1 Acoustic and visual techniques

Acoustic-based docking strategies are commonly found in underwater stations since sound waves prove reliable over medium to long distances. Docking with acoustic techniques proves difficult because, as the AUV gets closer to the station, the more feedback there must be. Due to inherent limitations of sound waves, such as low bandwidth, high latency, and low data rate, the expected level of accuracy is not met [2,10]. It is possible to have an idea through [11] that among 16 devices, the highest rate of data transfer is 150kbps over 120m, and the highest distance is 10km with a data transference of 5kbps. Other options are available but not only are they cost ineffective, but also consume more power than the standard devices [2]. In acoustic communication, a recent AUV development achieved an accuracy of $0.2m \pm 0.05\%$ up to $7m$, requiring 3 beacons so that the triangulation could be effectively accurate [12]. However, for a small-sized AUV (width $\leq 0.20m$) a higher accuracy might be needed.

Unlike acoustic techniques, visual-based strategies are less common, but their accuracy during the docking sequence makes this option appealing. These strategies are usually accompanied by the acoustic systems, making a hybrid between visual (for short-ranges) and acoustic (for long ranges) [13]. Visual sensing uses the on-board camera to identify light beacons and guide the AUV according to the location of these, using feature detection mechanisms. In [14] it is possible to

see an implementation of this method for a hovering-type underwater station, by following a simple set of vision-based algorithms to perform the docking sequence. Article [3] presents an implementation with three light beacons of distinct colors which can be identified by the AUV. As it has been tested in the previous work, this optical strategy has a remarkable accuracy of 2 millimeters in the horizontal coordinates and 15 millimeters in the vertical coordinate. Still, instead of simply using a visual technique which can only be applied to the docking maneuver in stable conditions, a short-range communication technique could not only aid the docking maneuver, but also implement a feedback mechanism.

2.1.2 Optical modems

Recent advances in AUV technology show that optical communication has potential over short to medium ranges. Articles [6,11] make a comparison between several types of communication possible underwater. Acoustic communication, as mentioned above, has limiting properties, such as the speed of sound, which is close to 1500m/s. Compared to light speed in water ($2,3*10^8$ m/s), acoustic lags in data transmission rate. Optical communication systems can transfer data in the orders of Mbit/s to Gbit/s, with a range of up to 200m in the best cases, as well as being very cost-efficient and low-power. Due to these values, data transmission on a real-time basis is possible, which could, for example, be used to control the AUV in the docking phase if the situation requires it.

In Table 2.1, a list of the most recent optical modems and their specifications is presented. Since cost-effective solutions are preferred, only light emitting diode (LED) based constructions are shown. It is important to note that as of October 2017, the price range of the components from Thorlabs are:

- Light emitting diode (LED) (405 - 680 nm): 6,45€ to 52€ [15].
- Avalanche photodiode (APD) (400 - 1000 nm): 995€ to 1.069€ [16].
- Photodetector (PD) (350 - 1100 nm): 12,50€ to 95,50€ [17].

Table 2.1: List of recent optical modems developed

#	Power (W)	Range (m)	Transmission rate (Mbit/s)	T/R Type	Aperture	Ref
1	12~24	10	0.16~1.6	LED/PD	12°	[18]
2	N/A	8	0.92	LED/PD	N/A	[19]
3	16,5	100	5	LED/APD	<30°	[20]
4	2~10	1,60~2,30	1	LED/PD	<30˚	[11]

Observing the table at first glance, it is possible to see a significant difference in range for device number 3. This is thanks to the usage of the APD, which is highly sensible. However, as seen before, the cost for these devices are high in comparison to regular PDs. Other devices have reasonable measurements, which appear sufficient for docking range. Device number 1 shows a higher range than 4, but in turn it has a smaller aperture size and requiring more power to transmit. Since there is more information available through device number 4, it will be used as an example to analyze these optical modems.

System analysis

For device number 4 from Table 2.1, two colors are used, blue and green, for bidirectional communication. The choice of colors was made based on Figure 2.1 (taken from [21]). The absorption is much less effective in the visible light spectrum, especially in the blue-green area.
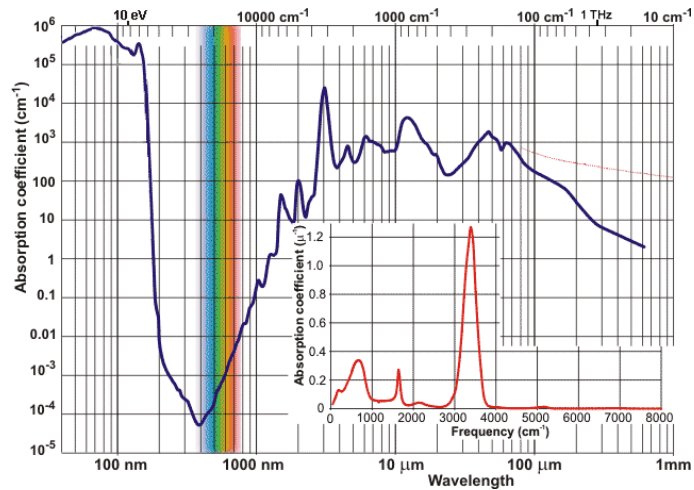
Figure 2.1: Water absorption spectrum

Considering the results observed in [11], for the same transmitted power, the radiation diagram follows the Figure 2.2 for the blue and green ends respectively:
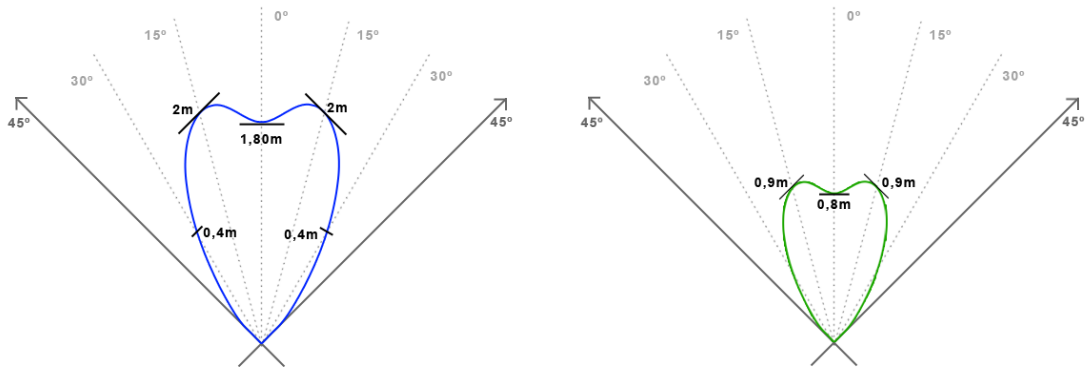
Figure 2.2: Radiation schemes for the respective blue and green ends

Observations can be made from this figure:

-   The variation of received power is very different for different angles, which requires different strategies to compensate for these changes.
-   The maximum range seen in blue and green ends differ in about 1m, which must be dealt with accordingly.

In Figure 2.2, the green end has a much smaller range than the blue end. This means that, for example, at the range of 1,50m for the conditions tested in [11], the blue end might be more reliable than its counterpart. This suggests different settings must be used for each color end. There are two factors to have in mind while thinking of a solution in terms of which end should each color be implemented:

-   The green end has less range for the same transmitted power as blue.
-   The blue end can get information across more easily due to lower absorption.

Optical devices that use bidirectional configurations are composed of a transmission (Tx) layer and a receiver (Rx) layer. The Tx layer is comprised by a LED light source and a MOSFET light source driver. The Rx layer uses a PD light detector to receive the signal, which is then amplified and filtered accordingly, followed by a comparator to implement the on-off keying (OOK) modulation. Constructions using LED/PD combos or similar should be implemented using the diagram from Figure 2.3.
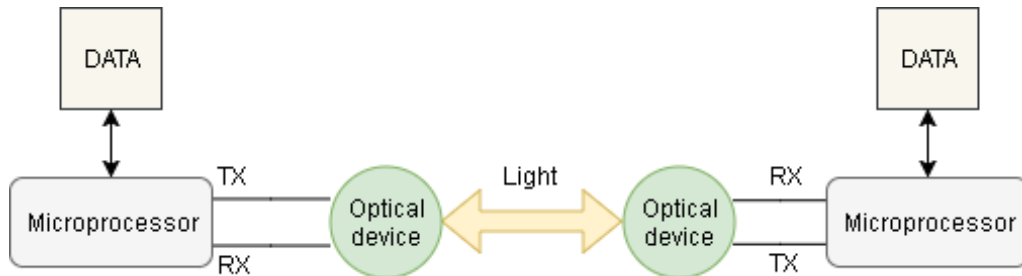
Figure 2.3: Overview of the optical system design

It is possible to observe from this diagram that optical devices interconnect to a microprocessor which processes data from a higher level, which in the case of the AUV end, connects to the core, and for the underwater station, it connects to the station's system (represented as DATA in the figure). As it was mentioned previously, to manage the underwater environment's harsh conditions, a robust communication protocol should be of concern as the optical devices are introduced. This work is done mostly at transport level on the microprocessor that processes the data.

In terms of connection, the optical devices are only comprised of Tx and Rx layers, as Figure 2.3 demonstrates. This design constraint limits the device choice from the microprocessor, since it requires an asynchronous communication with two wires. Being present in most devices, universal asynchronous transmitter-receiver (UART) serial communication is the most appealing for this task. As a final note, half-duplex communication should also be accounted for, as systems of lower cost tend to be unable to transmit at full-duplex [11,18].

2.2 Error-correction coding

Error-correction mechanisms are required to implement a robust communication channel. There are three types of error correction systems:

- Automatic repeat request (ARQ).
- Forward error correction (FEC).
- Hybrid ARQ.

From [6], it is possible to observe that underwater communication protocols use mostly either an ARQ system, or a FEC algorithm for error-control mechanisms. Therefore, to be able to assess which system is better for this type of channel, a study on these is required.

2.2.1 Automatic repeat request

ARQ is an error-control method in which every segment of data transmitted must be acknowledged by the receiver in a certain time window. This system is used to avoid losing data unnecessarily in channels of variable conditions. There are three types of ARQ systems [22]:

- Stop-and-wait.
- Go-back-N.
- Selective-repeat.

Assuming there are no timeouts, stop-and-wait is a type of ARQ system that waits for a response from the other side every time a transmission is sent. If an acknowledge (ACK) is received, the next transmission is sent. Otherwise, if a not-acknowledge (NAK) is received, it means that the information was corrupted, so the sender resends the packet. This is the simplest form of ARQ known.

Go-back-N is a special case of sliding window protocol, in which the sender keeps track of the sequence numbers of packets to be sent by a window x, and every time there is a request for retransmission (NAK), the sender goes back N packets sent (such that $N < x$), and retransmits all N packets again.

Selective-repeat is another special case of the sliding window protocol, in which the sender also keeps track of the sequence numbers of packets to be sent by a window x, but in this case, when a NAK is received, the sender only sends the packet that was not-acknowledged. This is an upgrade from Go-back-N, as only the packet N is sent, while the other needs to resend all N packets.

However, since full-duplex is required to be able to send packets without waiting for a response, Go-back-N and Selective-repeat are restricted to the stop-and-wait behavior in half-duplex communication.

2.2.2 Forward error correction

FEC is a technique used for correcting errors in transmissions over noisy communication channels. The main idea is that the transmitter encodes a message with redundant information so that the data can be recovered after being affected by errors.

These algorithms can be divided in two main categories:

- Convolutional codes.
- Block codes.

A study featuring these two types is shown below.[1]

Convolutional codes

Convolutional codes are a category of error-correcting algorithms that generate parity symbols through the convolution of a Boolean polynomial over the message to be encoded. Due to the nature of the polynomials used, these codes process data on a bit-by-bit basis. They are identified by having a constraint length K, where r parity bits are produced from the convolutional process. In this method, only the r bits are transmitted. One of the main advantages of convolutional coding is the use of soft decision decoding, where decoded bits are read as a probability of being 1 or 0. However, in situations where, for example, energy consumption needs to be avoided, then this feature is disadvantageous, as more data is needed to transmit information. In recent advances, Turbo codes and low-density parity-check (LDPC) codes are replacing old convolutional codes, due to their simplicity and efficiency [24]. These codes perform best when applied to parallel types of communication, where the bits that form a byte are sent separately, so it is more probable that single-bit errors occur.

Block codes

Block codes are the convolutional algorithms' counterpart, processing data in packs of bits (bytes) rather than bit-by-bit. The most basic example of these codes is block repetition, in which the data is merely repeated. Other examples are Hamming codes and other multidimensional parity check codes, which prove effective against burst errors, in contrast to convolutional codes [25,26]. These types of algorithms are more appealing for an underwater optical communication (UWOC) channel, where all data is sent through one source only, due to this channel using serial communication, where burst errors are more common than single-bit errors. This is because there is only one wire for byte transmission. Burst errors are also far more likely to happen in an UWOC link not only because the photodiodes may capture intense noise in shallow waters, but also high rapid variations may occur to the channel in busy locations. By means of [26], a study presenting

---

[1] Detailed description of the error-correcting mechanisms can be seen in "Error-Correction Coding and Decoding" [23].

comparison between low-density parity check (LDPC), Reed-Solomon (RS) and Bose–Chaudhuri–Hocquenghem (BCH) algorithms shows that RS vastly outperforms the other algorithms in cases where burst errors are more prominent. If processing power is not a problem, RS codes may achieve up to 50% more decoding than its peers for burst errors, possibly correcting a bit error rate (BER) of up to 1×10^{-2} in this case. The results show that there is an 8-dB improvement in SNR for a BER of 10^{-4}. Due to the nature of the channel and the reliability of RS codes, a study specific to these codes is conducted.

Reed-Solomon Coding

The encoding procedure assumes a packet of (N, K) symbols, where N is the total packet length, K is the data length, and the encoding symbols are then generated with a N-K rate. The (N, K) pair may also appear in the form of coding rates, with K/N. The combination of K words does not necessarily need to be in order to be able to reconstruct all the N words. The following process will be used to describe the encoding and decoding process of the Reed-Solomon error correction algorithm. The original procedure for the encoding technique of Reed-Solomon is described in [27].

A message is mapped as an array $x = [\, x1, x2, \dots, x_k\,] \in F^k$ to a polynomial p_x, being

$$p_x(a) = \sum_{i=1}^{k} x_i . a^{i-1}$$

Which then is evaluated x points, resulting in the following code:

$$C(x) = [p_x(a_1), p_x(a_2), \dots, p_x(a_n)]$$

$C(x)$ is a function that satisfies $C(x) = x.A$ for the following matrix A with elements from the field F^k

$$A = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & a_n^{k-2} \\ a_1^{k-1} & \cdots & a_n^{k-1} \end{bmatrix}$$

The encoded message is determined as A, and sent over the channel. It should also be noted that RS codes do not need to be standardized. Shortening of RS codes is a common technique to allow smaller sizes to be encoded. On the same note, a technique known as puncturing makes it possible to omit certain encoded parity symbols.

Although there is one way to encode the message in Reed-Solomon, the decoding procedure can be done in many ways. Still, the decoding process can be generally done in five steps:

1. *Syndrome calculation*. Using an algorithm such as Berlekamp-Massey it is possible to quickly detect if the message is corrupted to generate a warning before it decodes.
2. Taking the syndromes polynomial, the *erasure locator* takes place. The same algorithm locates the bytes that are in error.
3. The next step comprises of an evaluator, to address how much the message was corrupted, while analyzing if it is possible to decode the message.
4. Following the previous processes is the storage of the known errors in a magnitude polynomial, which is used to subtract the errors in the message to decode it successfully.

5. Repair the message by executing a subtraction of the previous polynomial with the message received.

RS codes can correct erasures (where symbol error locations are known) at a $t = N - K$ rate. This means that a certain erasure in a message can be corrected if at least one FEC symbol is present. Errors (where symbol error locations are unknown), however, can correct messages at a rate $^t/_2$.

As the RS coding algorithm is explained, the remaining issue is to know what is the best (N, K) pair. In most cases, the choice of (N, K) pair is done arbitrarily, left to the designer of the protocol to test and simulate the various possibilities [28–30]. In [28], a showcase of a pair (255,129) RS code was used in to test the improvement of an UWOC channel with a BER of magnitudes 10^{-4}, achieving an improvement in SNR of 8db. From [29], tests were conducted using RS(255,129) and RS(255,223). This shows a gain of 6 and 4 dB respectively. In contrast, articles such as [31,32] avoid the usage of such notations by implementing an adaptive coding based either on external functions or based on the data length. Article [32] presents a study with fixed coding rates, such as the ones mentioned above, *versus* adaptive coding rate for an underwater acoustic communication network. Results present three different links with different distances. The RS code rate changes to adapt this distance difference. For example, for the distance of 340m, the punctured RS code was 0.41, while the fixed code rate was 4/5. Comparison between these two schemes show that variable RS code rates are greater in terms of both efficiency and reliability.

2.2.3   Hybrid automatic repeat request

Hybrid ARQ is a system that uses both ARQ and FEC mechanisms [33]. Article [32] demonstrates how implementing this system in a very noisy channel can accomplish a reliable communication channel. Using Reed-Solomon as FEC coding, results show that for the channel with the most noise, packet success rate increases from 75% to ~90%. In general, there are two types of hybrid ARQ available.

Type I is the simplest way of implementing this system. It first adds a simple parity check to each transmission. If a NAK is received, then the retransmission adds error-correcting information to the packet.

Type II is a strategy that differs in the way retransmissions are performed. The first packet is transmitted with a simple parity checking code, just as type I. However, when a retransmission is requested, there are two ways of handling this.

The first is chase combining, where the same packet is sent with the same FEC information, but the receiver combines the packets to increase the chances of decoding. Take the packet to be sent in Figure 2.4 as an example, where the "1" and "2" are redundant bytes of RS.

Figure 2.4: Packet to be sent

At the receiver side, the packet arrives with the structure of Figure 2.5.

Figure 2.5: Packet received

As seen previously, two RS bytes added to a message may decode one error. Since the message has two errors, it cannot decode it, so a NAK is sent from the receiver side. The packet is then retransmitted. At the receiver side, once again, the packet that arrived is as seen in Figure 2.6.

Figure 2.6: Chase combining - Packet received after retransmission request

By combining the letters that do not contain errors, the packet can be successfully decoded, avoiding yet another retransmission.

The second way to handle retransmissions in type II hybrid ARQ is by incremental redundancy (IR). When there is a request for a retransmission from the receiver side, the sender does not send the packet with the same information, rather, it sends only FEC coding to be appended to the previously transmitted packet.

Taking the example seen from Figure 2.4 and the packet received at the receiver side in Figure 2.5, the receiver sends a NAK in the same manner. However, differing from the chase combining strategy, the retransmission is as shown in Figure 2.7

Figure 2.7: Incremental Redundancy - Packet sent after retransmission request

As it can be seen, only FEC symbols are sent. As this gets appended to the previous packet, it is now able to decode successfully. In this case, the packet sent is much smaller than the previous case, saving energy and increasing transmission speed.

Nevertheless, the creation of a hybrid ARQ system implies a new problem to be solved: variable FEC coding. This means, for example, that packet number 1 will have a pair of RS(64, 38) and packet number 2 will be sent with RS(64, 40). Supposing that packet number 1 is sent at a given moment, and that the receiver is configured by default to accept a packet with 12 parity bytes, then it will not be able to receive a packet with 10 FEC bytes without an additional strategy. Since the knowledge on how many additional FEC bytes is required on both the transmitter and receiver sides, it should be natural that a packet design containing this information is essential. There are two ways of achieving this result.

Packet encapsulation

Packet encapsulation consists in delimiting the packet with a character such that when it gets captured by the receiver, it represents the end of a packet. This means that the character used for delimitation must not be used either in the message or in the FEC coding.

In [34], high-level data link control (HDLC) uses bit stuffing for packet encapsulation. Using the flag sequence 01111110, it marks the start and ending of each packet. To make sure this sequence does not appear in the data itself, it encodes the information in a way that whenever five adjacent ones are observed, it follows with a zero. This way, there are no six consecutive ones in the message. The decoding process reverses this operation: whenever a zero follows five ones, the zero is deleted, and when six ones are observed, it is interpreted as a special framing sequence. The process of increasing data in a binary fashion for the sake of delimiting packet is called "bit stuffing" and increases the data size. In the worst case, HDLC framing can add 20% of overhead to the data itself.

As opposed to bit stuffing, point-to-point protocol (PPP) uses a byte stuffing technique [35]. It uses the same binary sequence as HDLC for packet encapsulation (01111110). The algorithm restricts this symbol by replacing values of 0x7E (01111110) with two bytes: 0x7D 0x5E. In the data, appearances of 0x7D are replaced by 0x7D 0x5D. The same way as HDLC, the decoding process performs the reverse operation: whenever it sees the 0x7D, it deletes that byte and executes a XOR operation with 0x20 and the next byte to obtain the original sequence. As the worst-case scenario for this protocol, it can double the message length.

From [36], consistent overhead byte stuffing (COBS) shows a comparison between these two algorithms, while introducing a new one. COBS takes the data up until 255 bytes as input and outputs a message in the range [1,255] by eliminating all zeros. The zero can then be used to delimitate packets. The overhead of the worst-case scenario for this algorithm is merely one byte.

This expected overhead makes this algorithm appealing *versus* its peers, however, a problem is of concern with these encapsulation alternatives. If, for example, COBS was chosen to be implemented, then both the data and FEC bytes would have to be encapsulated. In a very noisy channel, the byte that is left because of packet encapsulation could be affected, compromising the whole packet.

Header

A header is a portion of the packet that precedes its body and contains additional information. Analyzing [31,32,37], using a header to evaluate the packet's structure is the most used strategy among hybrid ARQ schemes. A header containing the sequence number, the number of FEC bytes and the number of data bytes is added to let the receiver know what the size of each partition is. The overhead of this setup depends on the number of bytes added to the header, as it must have its own FEC partition to avoid packet loss on the header itself. As reviewed before, using RS as FEC, then in a header of 3 bytes, 6 additional bytes are required to correct errors in all three bytes. The total overhead in a packet then becomes then 3+6=9 bytes. Comparing these strategies, the header alternative appears to be more appealing, as it is more robust against using a delimitation that is susceptible to corruption. However, the design must be studied to reduce overhead cost.

2.3 Conclusion

This chapter reviewed the underwater docking strategies which are used currently, while analyzing the recent technology and how it can benefit the docking maneuver.

Following this study, the error-correction coding techniques were researched in order to implement a robust version of the optical communication technology.

It is now possible to use the reviewed literature to effectively design the protocol intended.

Chapter 3 - Material and method

The implementation method is shown below, divided into the following sections: requirements, system architecture, material, proposed solution, proof of concept, and finally simulation method.

3.1 Requirements

Implementing a robust communication system requires additional coding for the exchange of information, either by parity checking, either by error correction algorithms. ARQ systems are also known to be extremely reliable as it is simply a much-needed feedback mechanism on the communication channel. With these options available, there was a need to establish the boundaries of the proposed solution. Below is a list of the requirements set for this dissertation:

- Maximum efficiency: adding ARQ responses and additional FEC drives the power consumption up as more bytes are sent, which hinders efficiency.
- Maximum reliability: on a channel where BER is irregular, there are cases where a very reliable communication is required.
- Controllable: full control over the AUV is desirable since the system will also be used to override the AUV's automatic docking scheme.
- Half-duplex communication system: the system should implement a mechanism to avoid simultaneous transmissions to be applied in most devices of lower cost.
- System testing: the proposed solution should be simulated with different case-scenarios to test both reliability and efficiency.

For the maximum efficiency and maximum reliability requirements, a means for comparison must be established. Since the usage of fixed code rates is common in underwater communication [32], comparisons will include these designs.

3.2 System architecture

The system architecture presents the core of the structure and which algorithms were used. As it was stated in the requirements, half-duplex communication was needed. For this reason, a master-slave architecture was proposed. In this type of design, the slave side is always responding to the other, and cannot act on its own, while the master end is the one sending commands and making requests. Adhering to the controllable requirement, full control over the AUV was also desirable, so having the master sit at the station's side ensures this feature. This implies that the AUV will not be able to establish communication on its own and can only respond to requests made from the station. This proposal fixed upcoming problems such as the issue of having the AUV and the station trying to establish connection simultaneously, adhering to the half-duplex requirement.

In the case of the devices from [11], the blue end should be implemented in the station, as the extensive range of the blue device would be cut down to the green device's range if it were implemented in the AUV side as seen in Figure 3.1, since as a slave it cannot initiate any communication without the master's request.
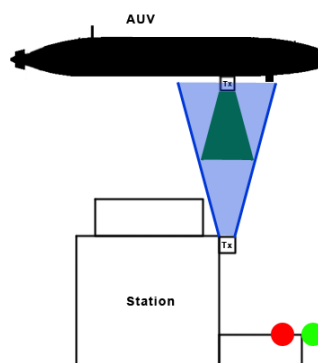
Figure 3.1: Example case of master establishing connection while the slave does not

Another observation that was made from the optical devices is that they require different approaches according to the relative location between the two ends. As means of demonstration, it is fair to mention that the maximum power output is in the angles of 0º to 15º. Taking the value of 15º for the blue radiation scheme as an example, the distance covered by the device at 1,80m is about $\tan(15º) \times 1{,}80m \times 2 \cong 0{,}96m$, while at short distances, for example at 0,3m it covers $\tan(15º) \times 0{,}3m \times 2 \cong 0{,}16m$, as it can be seen in Figure 3.2. This creates two different situations in which the AUV might be in.
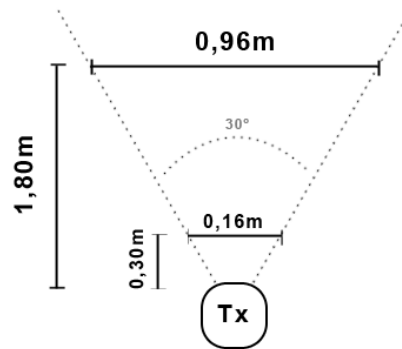
Figure 3.2: Device coverage at different ranges

- Far field: the distance covered by the optical device is large enough to tolerate localization errors, so the most evident problem is the low power which creates errors in the messages.
- Near field: the distance covered is now smaller, where misalignment issues are the most evident problem. This translates to a behavior where there is communication without errors due to high power reception, then suddenly no reception at all.

This suggested that using a system that is reliable to both noise and transmission failures was a good starting point as the system architecture. Reviews from literature show that both FEC coding and ARQ systems are respectively solutions to these problems. For this reason, the chosen error-correcting mechanism was hybrid ARQ, as the ARQ system handles efficiently the case where timeouts may occur in the near field and FEC coding handles the case where low SNR occurs in the far field. The type of hybrid ARQ scheme used is the type II with incremental redundancy, as the reduction of energy consumption in retransmissions adheres to the maximum efficiency requirement. As the type of ARQ system, stop-and-wait was used, to abide to the half-duplex requirement. The FEC coding chosen was Reed-Solomon, which proves to be very reliable in cases where burst errors occur, which is the case for UWOC. However, these choices required manipulation of the transmitted data so that it can withstand the harshness of the underwater conditions.

3.3 Material

Two Raspberry Pi 2, Model B (RPi2) were used to simulate the UWOC channel [38]. The most important feature that was searched for was the integrated UART serial interface (device with the name "ttyAMA0"), since, as it was previously mentioned in the literature, it can be used to implement the optical communication devices. A Linux-based operating system named Raspbian (version 4.9) was installed on these microprocessors.

Physically, the setup consists of two wires connecting to the UART pins of the RPi2s, and one wire for the common ground, Figure 3.3. This creates an ideal communication channel which can then be used to recreate a controlled version of the UWOC channel via software. A PC is connected to both RPi2s through a switch using ethernet cables.
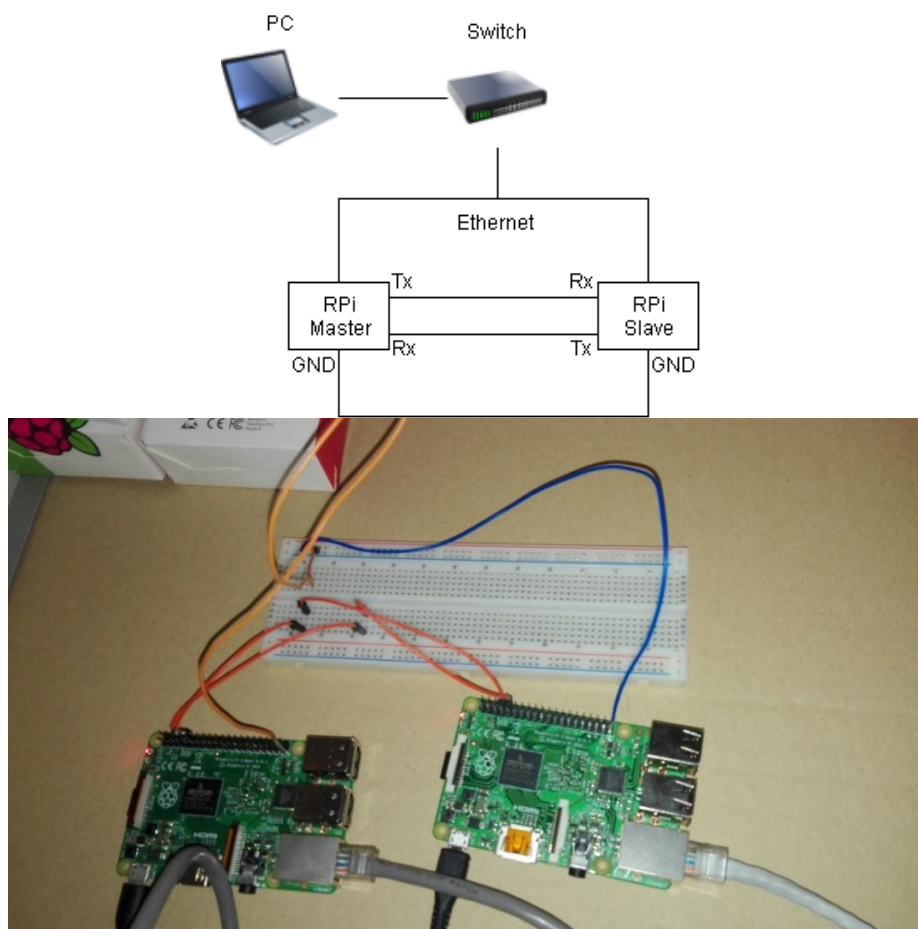
Figure 3.3: Setup for simulations – diagram and respective photo

The protocol was written in Python (version 2.7), with the following external libraries being used:

- A pure-python Reed Solomon encoder/decoder [39].
- Python serial port access library [40].

Besides these libraries, Matplotlib [41] was used as a simulation tool.

The final program for the master and the slave ends are displayed in the appendix. License for the external libraries are annexed.

3.4 Proposed solution

A detailed study featuring the design of the system had to be made before the implementation and conception of the protocol, an essential part of software creation. Firstly, packet design took place, explaining how the information is manipulated before a transmission. It followed with an investigation of the channel model for UWOC, to apply the error-correction algorithms effectively. Afterwards, the implementation of FEC coding was planned. Then the ARQ system was described in detail. Finally, as the strategy was ready to be employed, diagrams for the main functions of the protocol were made.

3.4.1 Packet design

A packet is a method of grouping data in a way that can be transported safely in a network. The packets that transport the data in this protocol had to have flexibility as its core attribute, since variable FEC coding was implemented with the introduction of the hybrid ARQ system. Packet length was then left variable to achieve this flexibility. This was only possible using either packet encapsulation or a header, as seen in the literature. The header proved to be more effective against its peers in a channel where SNR values might be low, as it does not use specific characters that can be easily corrupted during the transmission. The packet design considered contains the following information: header, header FEC, sequence number, data, and data FEC. The header had to contain the minimum amount of information, as it presents a constant overhead in the transmission. For this reason, only the information on packet length is transmitted. This way, the header itself became 1-byte long. With only this information, extra measures had to be taken.

From the receiver side, as the header is decoded, the only information available is the packet length. With this, taking the packet length a as an equation

$$a = 1 + x + y, \qquad (1)$$

where there is a constant of 1 indicating the sequence number, x the number of data, and y the number of FEC symbols. As a solution to such a problem, it was possible to standardize lengths to be able to make the various sections identifiable. Since the usage of UWOC should accept all kinds of data (from short commands to video requests), it was limited to segments of 8, 64 and 236 symbols.

The sequence number is a symbol that contains additional information and is appended to the data segment. It serves two purposes:
- Avoid mixing packets that were already decoded by alternating a sequence number.
- Indicate the end of a transmission.

In the end of a transmission, the final packet does not always fit the data that needs to be sent. This happens because the smallest data segment that can be sent is 8 bytes. For this motive, the packet needs to pad the missing symbols with arbitrary bytes to be recognized by the receiver.

This sequence number contains the number of symbols that are padded so the receiver can remove them. If it is not the final packet sent, then the sequence byte alternates between two numbers. In the end, the sequence number reserves the integers from 0 to 7 for the final packet, and alternate between two arbitrary numbers in the middle of a transmission. Integers 8 and 9 were chosen.

Variable y, being the number of FEC bytes, is added according to the data length. For 8 bytes of data (+1 sequence byte), the maximum number of FEC bytes caps at 18 symbols, since in the worst-case situation it can correct 9 errors. For reference purposes, below is a table containing the decoding rates for each packet, with a maximum of 18 FEC bytes:

Table 3.1: Decoding rates for each packet size

Packet size	Decoding rate
8+1	$\dfrac{18/2}{8+1} \times 100 \cong 100\%$
64+1	$\dfrac{18/2}{64+1} \times 100 \cong 13{,}8\%$
236+1	$\dfrac{18/2}{236+1} \times 100 \cong 3{,}5\%$

As an example, using the equation in 5 as a model for the incoming packets, the following procedure takes place for a packet that arrived with length 15 bytes:

1st: $15 < 64 \ => \ x = 8 + 1$

2nd: $y = 15 - (8+1) = 6$

So,

$$1 + 8 + 6 = 15$$

For now, the header, data, sequence byte and data FEC have been handled. What is left is to know how to protect the header against errors, which is the header FEC. Looking at the packet, the worst-case scenario is using the packet with 9 bytes with an additional 18 bytes. As such, the relation of the data to FEC is $\frac{9}{18} = \frac{1}{2}$ in this case. So, the header was designed to implement 2 additional FEC symbols.

With this, the overall packet design for the communication is shown in Figure 3.4.
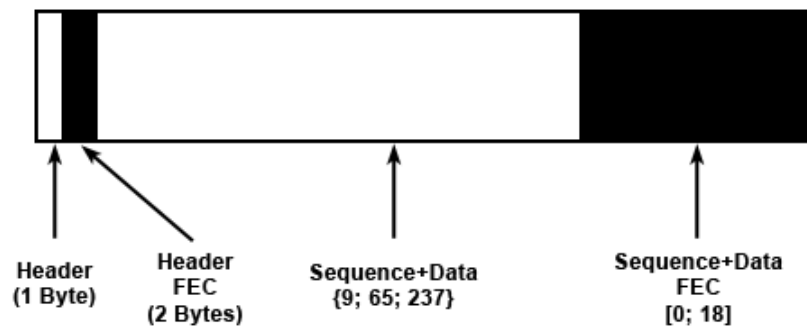
Figure 3.4: Overview of the packet design

Since the data transmission is handled reliably with this packet design, it then was necessary to characterize the communication channel in a way that can handle the addition of FEC efficiently.

3.4.2   Channel modeling

The communication link had to be modeled in order to apply a coding strategy that suits the underwater channel. According to [42], the closest noise model for an UWOC is additive white gaussian noise (AWGN) channel. However, being a model that describes events on a specific environment and the properties of an optical device, it was avoided, so that the protocol could be generalized. A binary symmetric channel (BSC) model was used instead, as they prove to be effective with Reed-Solomon's error-correcting characteristics [32]. For this motive, the symbol error rate (SER)was the variable tested. In a BSC model, the probability of symbol error is given by

$$P_s = \sum_{i=1}^{m} \binom{m}{i} P_b^i (1 - P_b)^{m-i}, \qquad (2)$$

where $m = log_2(L + 1)$ with L being the number of bits in a symbol. Taking the principles behind this equation, for a pair (N, K) of RS-encoded packets, the probability of not decoding a packet can be expressed as

$$P_{ndec} = \sum_{k=\frac{N-K}{2}+1}^{N} \binom{N}{k} P_s^k (1 - P_s)^{N-k}, \qquad (3)$$

where k is the maximum number of symbols that can be corrected.

With the model of the communication channel studied, the application of the error-correction algorithms had to be considered.

3.4.3 Error-correction systems

The error-correction systems discussed in the literature required additional planning, as they needed to be fine-tuned. Three implementations are presented: incremental redundancy, adaptive error-correction, and an error-correction control mechanism which was proposed.

Incremental redundancy mechanism

Hybrid ARQ uses incremental redundancy as a retransmission mechanism, which was discussed in the error-correction coding section of the literature. This means that every retransmission made contains only additional FEC coding symbols. Reed-Solomon features a puncturing technique which allows a part of the encoded symbols to be omitted during a transmission. For simplification purposes, take the example pair (13,9). Firstly, the packet is pre-processed with (27,9) as shown in Figure 3.5.

| A | B | C | D | E | F | G | H | I | 1 | 2 | 3 | 4 | 5 | 6 | . . . |

Figure 3.5: Packet encoded with maximum 18 FEC symbols

Secondly, the intended transmission is buffered by taking only the first 13 bytes and afterwards sent, as seen in Figure 3.6.

| A | B | C | D | E | F | G | H | I | 1 | 2 | 3 | 4 |
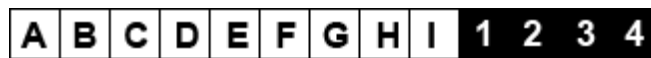
Figure 3.6: Packet with punctured FEC symbols

At the receiver side, the missing encoder symbols are replaced with arbitrary symbols so that the pair becomes (27,9) again. At the decoder, the padded symbols in the packet are marked as erasures, representing errors whose locations are known, Figure 3.7.

| A | B | C | D | E | F | G | H | I | 1 | 2 | 3 | 4 | X | X | . . . |

Marked erasures

Figure 3.7: Packet padded with arbitrary X symbols

This allows the algorithm to "correct" the padded bytes at a rate $t = N - K$, as opposed to errors which are corrected at a rate $\frac{t}{2}$. As such, with the missing FEC bytes "corrected," the algorithm can then correct the rest of the message as intended.

Adaptive error-correction

Based on the literature [32], the choice of (N, K) pair was left variable, which was possible by shortening the RS code. This is conceivable through the camera that's used to record missions and in certain cases be used to guide the AUV to the docking station. With this, it is plausible to assess

the relative height to the underwater station, feeding it to the protocol [43]. This was seen as an assumption, as it uses other modules of the AUV available. By implementing a map of the locations where the AUV might be in, along with the SER for each location, it is possible to retrieve them at a later stage to create a flexible (N, K) pair. This way, the FEC acts according to the location, avoiding standardized notations which might not be of use in all situations. This, however, is not directly accessible by the station modem end so it needs to request this information from the AUV. Since this error map relies on a SER that is associated to a location, it is highly reliant on the environment it is in. This means that for each unique environment, the symbol error rate for each location must be calculated beforehand. Finally, the relationship between the SER and the optimum coding rate is an issue that remains.

Assuming that the channel's conditions do not change during a packet transmission, the probability of the receiver decoding a packet is given by equation (3) from channel modeling. From the requirements, priority to either decoding capability or efficiency had to be given. As the objective of this dissertation is to apply a reliable communication protocol, reliability was favored. Firstly, packets of higher lengths had to be limited to high decoding chances. Since the probability of not decoding a packet never reaches 0 practically, packets of 64+1 and 236+1 were applied if $P_{ndec} <$ 0.005 for a maximum of 18 FEC symbols. However, equation (3) could not be used for packets of 236+1, as the number of symbols is too large to process in a binomial cumulative distribution function. An approximation to normal cumulative distribution was done instead with the following equation [44]

$$P_{ndec} = 1 - \frac{1}{2}\left[1 + erf\left(\frac{x}{\sqrt{2}}\right)\right], \quad (4)$$

where erf is the error function and x is given by

$$x = \frac{k - c - \mu}{\sigma}, \quad (5)$$

with $k = \frac{N-K}{2} + 1$, $\mu = \sqrt{NP_s}$, $\sigma = \sqrt{NP_s(1 - NP_s)}$ and $c = 0.5$ (continuity correction).

Iterations of these two equations for a value of $P_{ndec} < 0.005$ showed that for 237-sized packets, $P_{ndec}^{237} = 0.00499$ with $P_s^{237} = 0.0179$, and for 65-sized packets, $P_{ndec}^{65} = 0.004957$ with $P_s^{65} = 0.0596$.

Since the larger packets were delineated to very low error decoding probabilities, reliability is ensured. At this moment, only efficiency needed to be ensured. Given a K/N code rate, efficiency at a probability P_s can be expressed as

$$H = \frac{K}{N}(1 - P_{ndec}(N, K)), \quad (6)$$

With this, the code rate K/N should be maximizing H. For a data size K, the optimum packet size N with $N - K = 2t$ is given by

$$H = argmax\left[\frac{K + 2t}{N}(1 - P_{ndec}(N, K))\right]_{t=1}^{t=\frac{18}{2}}, t \in \mathbb{N}, \quad (7)$$

The relationship between a code rate K/N and symbol error probability P_s was then established.

Although the adaptive error-correction is a mechanism which chooses the best (N, K) pair, it relies on the accuracy of the AUV location, which might not be always correct.

Proposal of error-correction control

As mentioned in the literature, acoustic-based localization techniques at close-range are not always reliable. Despite aiding the communication protocol, relying solely on the locations the AUV provides can be detrimental for efficiency. As such, an error-correction control mechanism that can assess the quality of the communication from a higher level was proposed. Using the ARQ system that is part of the design to provide quality control can increase not only efficiency, but reliability as well. A global circular buffer was created to store information from the ARQ system as shown in Figure 3.8.
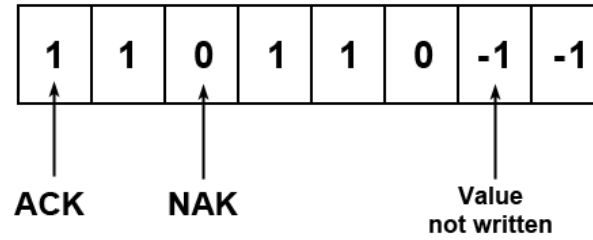

Figure 3.8: Buffer design

With the scheme of the buffer created, only the controlling method was needed. Taking equation (3) from channel modeling and equation (4) from adaptive error-correction, it was possible to assess the probability of a retransmission for a given (N, K) pair, as it is the same as P_{ndec}. This implies that for a specified window n, the amount of first order ACKs *vs* NAKs is

$$\frac{NAKs}{ACKs + NAKs} = P_{ndec} \cdot n, \qquad (8)$$

This means that for a certain P_s that the AUV feeds the protocol, a related P_{ndec} such that N-K maximizes efficiency (refer to adaptive error-correction) is calculated. However, if there is a discrepancy between the SER given by the AUV and the real P_s, then according to the proposal, the associated P_{ndec} also changes.

With this, the only issue remaining is the choice of the window n. If n is too small, then the chance of the controller changing the code rate wrongly increases. Yet, if the n is too large, then it will take a long time to act when there is a discrepancy of probabilities. For this reason, a test was conducted to understand what the experimental error is for each choice. An arbitrary point $P_s = 0.08$ was chosen to be tested. From equation (7), the pair (N, K) associated is (11, 9), with $P_{ndec} = 0.218$. The code used to simulate these values can be seen in the appendix. Table 3.2 is the test results, where 100 trials were made for each n shown.

Table 3.2: Experimental error test – 100 trials for each n

Number of packets n	Max difference max$\|$E-T$\|$	Experimental error $\|$E-T$\|/n$ (%)
10	3.82	38.2
100	11.19	11.2
1000	32.90	3.3
10000	112.99	1.1

In this table, E is the number of experimental packets decoded for each n and T is the theoretical value calculated using equation (3) with $P_{ndec} \cdot n$. This suggests that not only is a window n required for the FEC control, there must also be a variable that implements a tolerance according to the n used.

At this point it was possible to send data in a variable packet with variable FEC bytes, as well as a being able to control of the (N, K) pair according to the number of retransmissions made. However, the way that the protocol handles the ARQ system is an issue that is still unresolved. The next part will describe how the system handles the usage of ACKs, NAKs, and timeouts.

3.4.4 Automatic repeat request system

As part of the system architecture, an ARQ stop-and-wait system was also implemented. This part presents how the ARQ protocol was employed. Figure 3.9 displays a representation of how the communication channel reacts to the reception of an ACK, as well as what happens when the master (station) requests information from the slave (AUV). In this case, a packet containing a command (shown as CMD) is sent from the master side. Upon arrival at the slave end, it decodes this packet and processes it. Since it is a command that requests information, no ACK is sent back, as it is replaced by a packet containing data. This sets the master in receive mode, which is now the one sending ARQ responses to the slave.
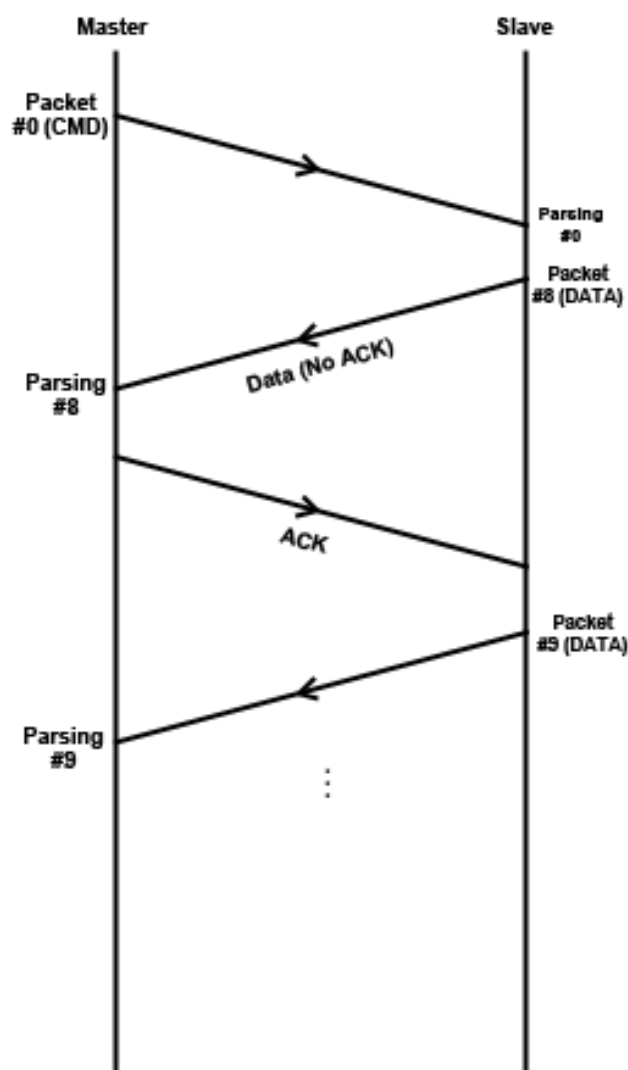
Figure 3.9: ACK in a data request command

Figure 3.10 represents a similar representation for a NAK reception. As it has been mentioned before, a NAK is fruit of undecodable packet at the reception and requests more FEC bytes to be sent. It should be noted that a retransmission after a NAK is only made if the number of FEC

symbols does not reach its maximum, because in this case it only spends more energy to transmit information, decreasing efficiency.
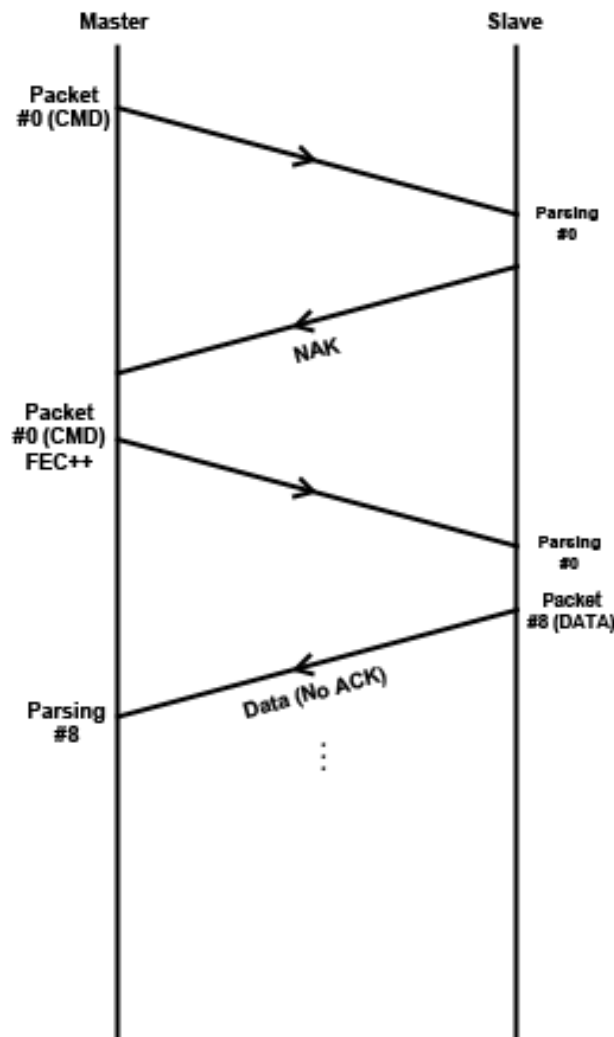
Figure 3.10: NAK in a data request command

Figure 3.11, likewise, characterizes the timeout version of the ARQ system. Timeouts represent two situations:

- The packet sent is lost.
- The ARQ response sent after the receiving end gets the packet is lost.

From the figure, the first transmission shows a failure in transmission from the packet. Here, the slave does not receive it so only the master will timeout. From the perspective of the slave, nothing has happened yet. The second transmission reaches the slave, but here, the ACK or NAK fails to arrive at the master, starting both timeout counters. The timeout at the slave is not triggered since it starts its countdown at a later stage and gives enough room to receive the retransmission from the master.
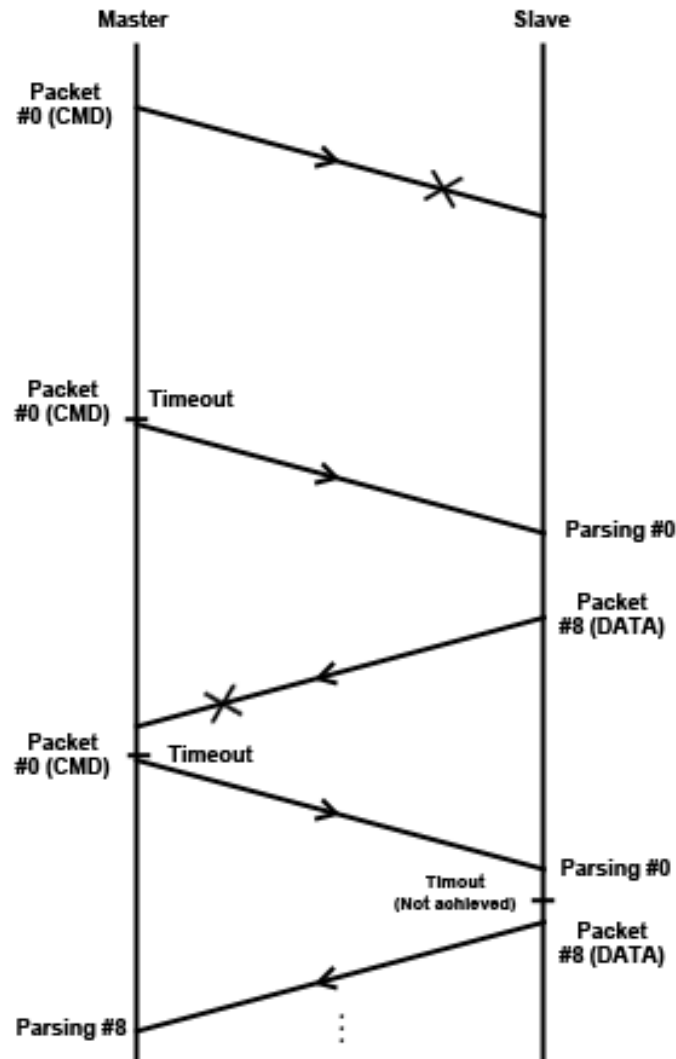
Figure 3.11: Dealing with a timeout in a data request command

Following these schemes, the only issue left to resolve was the situation where the transmission has ended. The connection is over when there is no more information sent over the channel. Still, there is no exact way to know if the last ACK or NAK of a transmission is received successfully by the transmitting end. This means that a function that implements a final timeout counter had to be added, so that the receiving end may resend the ACK or NAK if the other end did not get it, as shown in Figure 3.12.
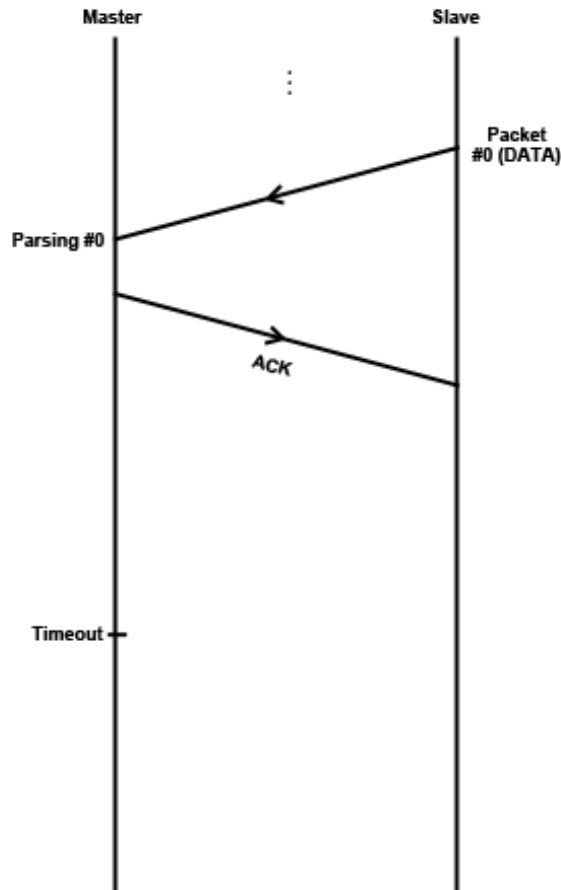
Figure 3.12: Final ACK timeout

In the next part, the design of the ACKs and NAKs that were portrayed in the ARQ scheme are presented.

ARQ Packet design

If the packets and headers sent are controlled by the number of FEC bytes, then ACKs and NAKs follow a similar structure. The ARQ packet design implements the same logic as the header, as seen in Figure 3.13.
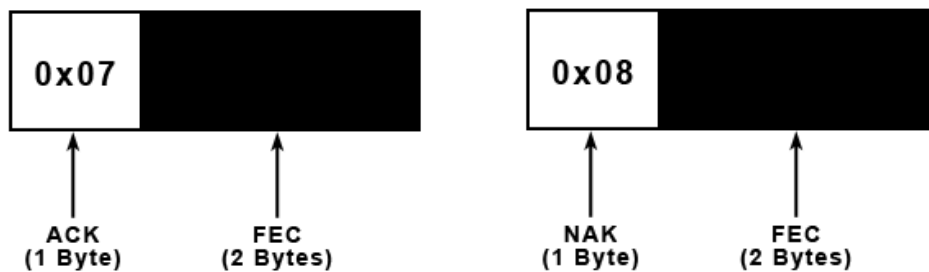
Figure 3.13: ACK/NAK Packets

It should also be stated that the choice of symbols for the ACK and NAK were made based on the condition that the header of a packet is always above 9 bytes, since it is the minimum packet length. This selection prevents mixing headers with ARQ responses.

3.4.5 Code design

The code design is a planning of the program in the form of diagrams (flowcharts). It encompasses 4 parts: the master: main(), master: send(), master: get_data(), and slave: main().

Master: main()

The main function is the first step ran when executing the program, Figure 3.14. It emulates a command-line terminal that infinitely receives inputted data and processes them for testing purposes. The inputs are treated in two steps: the first is used to retrieve the AUV location and set the FEC according to the SER. The second transmits the intended data. If a request for data was issued, then the final transmission is a packet instead of an ACK, as stated in the automatic repeat request sub-chapter. This packet received is then passed to get_data() function, which switches the mode to receiver.
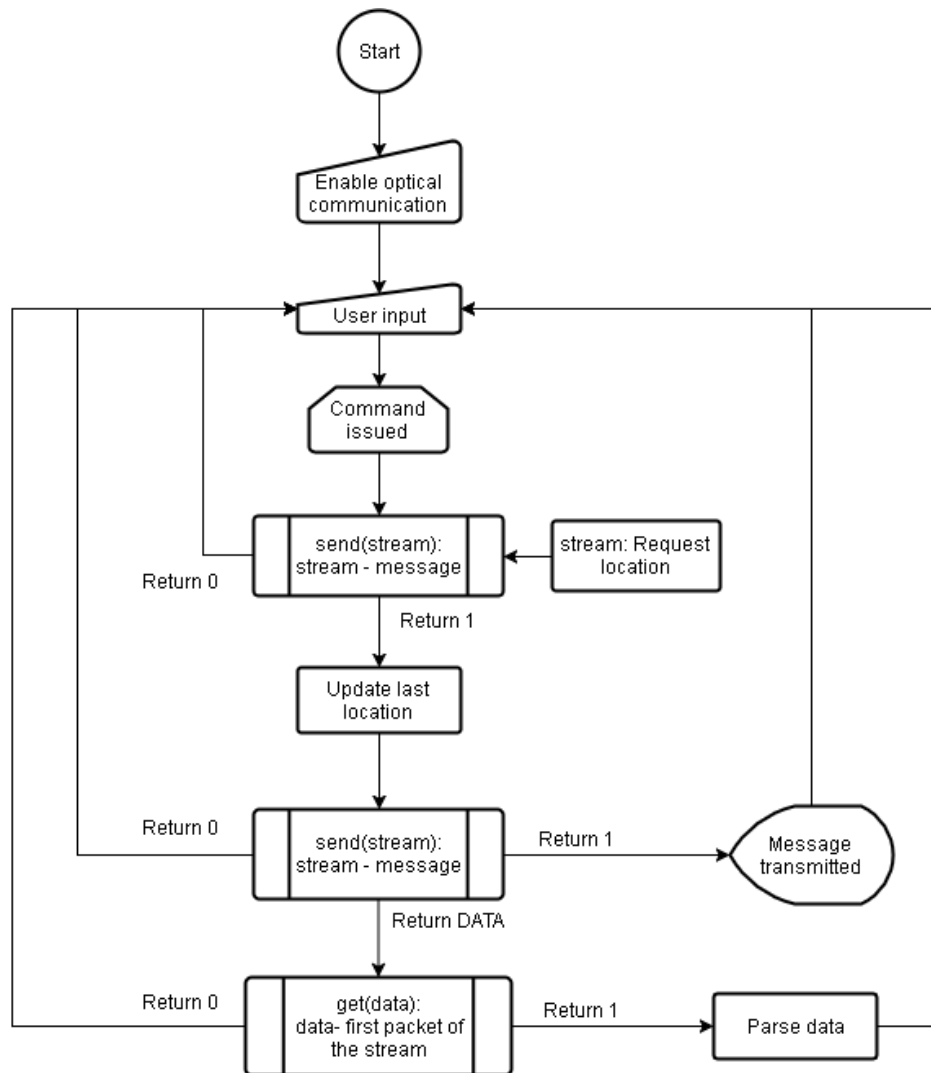
Figure 3.14: Main function – Master

Slave: main()

The main function of the slave end implements the reverse process of the master's main() function, Figure 3.15. It listens indefinitely for packets, parsing data the same way as get_data() function from master. If the message sent from master contains a request, then the slave enters transmission mode, where it sends the requested data.

Figure 3.15: Main function – Slave

send()

The send function implements the code to transmit packets to the other end, Figure 3.16. It starts by getting the SER associated with the location, which is declared in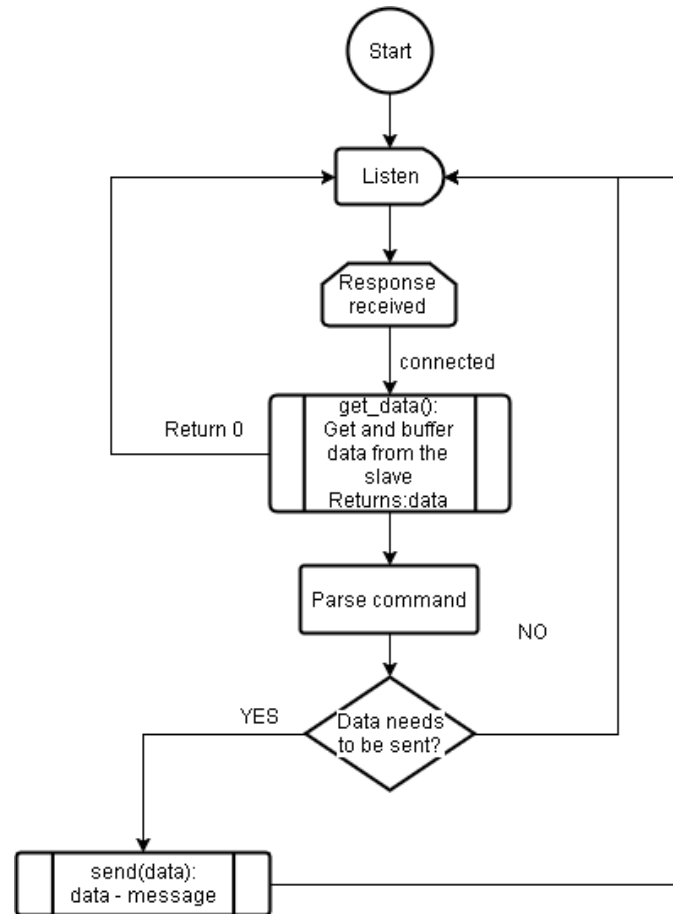 a structured file. For simulation purposes, the locations were divided in lengths of 10cm of range and angles [0,20] and]20,40], however, it needs to be fine-tuned when implemented on a real case-scenario. If it is the first use, then the location is yet unavailable for the master side. The minimum 2 bytes are then added to the message, relying on the ARQ system to manipulate the required FEC symbols. Otherwise, if the location is available, then the associated SER is used to optimize the FEC, while also being measured by the error-correction control (refer to error-correction systems).

The next step comprises of segmenting the data according to both the SER and data length. If a file that is larger than 236 bytes is sent, then it is divided in packets of 236+1 only if the location of the AUV allows it. Otherwise smaller packets are used instead.

As the sequence number gets appended to the packet, it is alternated if it is not the last transmission, otherwise it contains the number of padded symbols as mentioned in packet design.

Afterwards, the packet is encoded with maximum 18 FEC bytes, which can then be manipulated according to the location. As the incremental redundancy scheme describes, the packet must be punctured if the data and encoded symbols are to be sent. Otherwise, if it is a retransmission, only FEC bytes are sent, using the header as an alternating sequence number. This is done by using the integers below 7 which are free. With this, the retransmission system is also reliable against timeouts.

After the packet is sent, the program waits for an ARQ response coming from the other end. If an ACK is received, the program writes to the global circular buffer (refer to error-correction systems) a 1 and sends the next packet available. If a NAK is received, it is registered as a 0 in the buffer and it increases a local count. This count is used to select the FEC bytes to be sent after a NAK. The retransmission is discontinued if the number of FEC bytes has reached the maximum for the current packet, as mentioned in the automatic repeat request system. However, if the sender cannot read a response until the timeout, then the same packet is retransmitted while increasing timeout count, canceling the transmission if it reaches the maximum. In cases where the packet sent was a request, the acknowledge comes in form of data. This is detected if the header is a number above 9. Otherwise, if an ACK is received, the function repeats the process if there is more data to be sent or end the transmission.
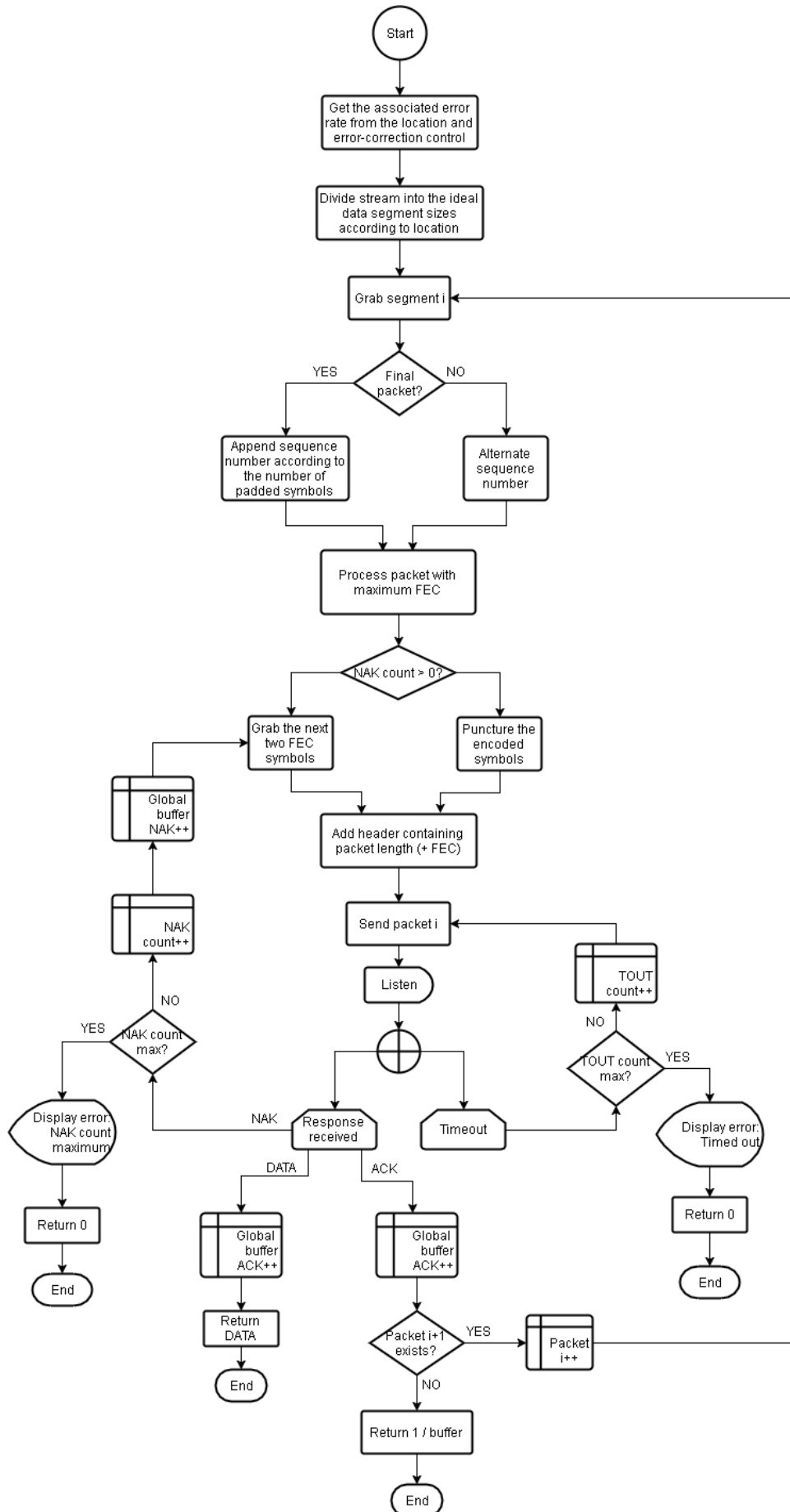
Figure 3.16: Send function - Master

get_data()

The get_data() function implements the reverse action from the send(). It listens to packets from the other end, processes them, and then sends a response, Figure 3.17.

Based on the header, the function reads the packet from the serial buffer and tries to decode it. If it is undecodable, an internal NAK count is increased. On the same note, if the NAK count reached the maximum, it should terminate the connection based on the automatic repeat request system's ending timeout scheme. Otherwise, if the packet is decoded successfully, then the sequence number is checked to understand if the packet was already decoded.

It then proceeds to remove the padded symbols if it is the last packet, append the decoded data to a buffer and sends an ACK. However, if it is not the last packet, then the function should wait for another transmission, which starts with the header. The transmission is discontinued if the timeout is triggered at any point.
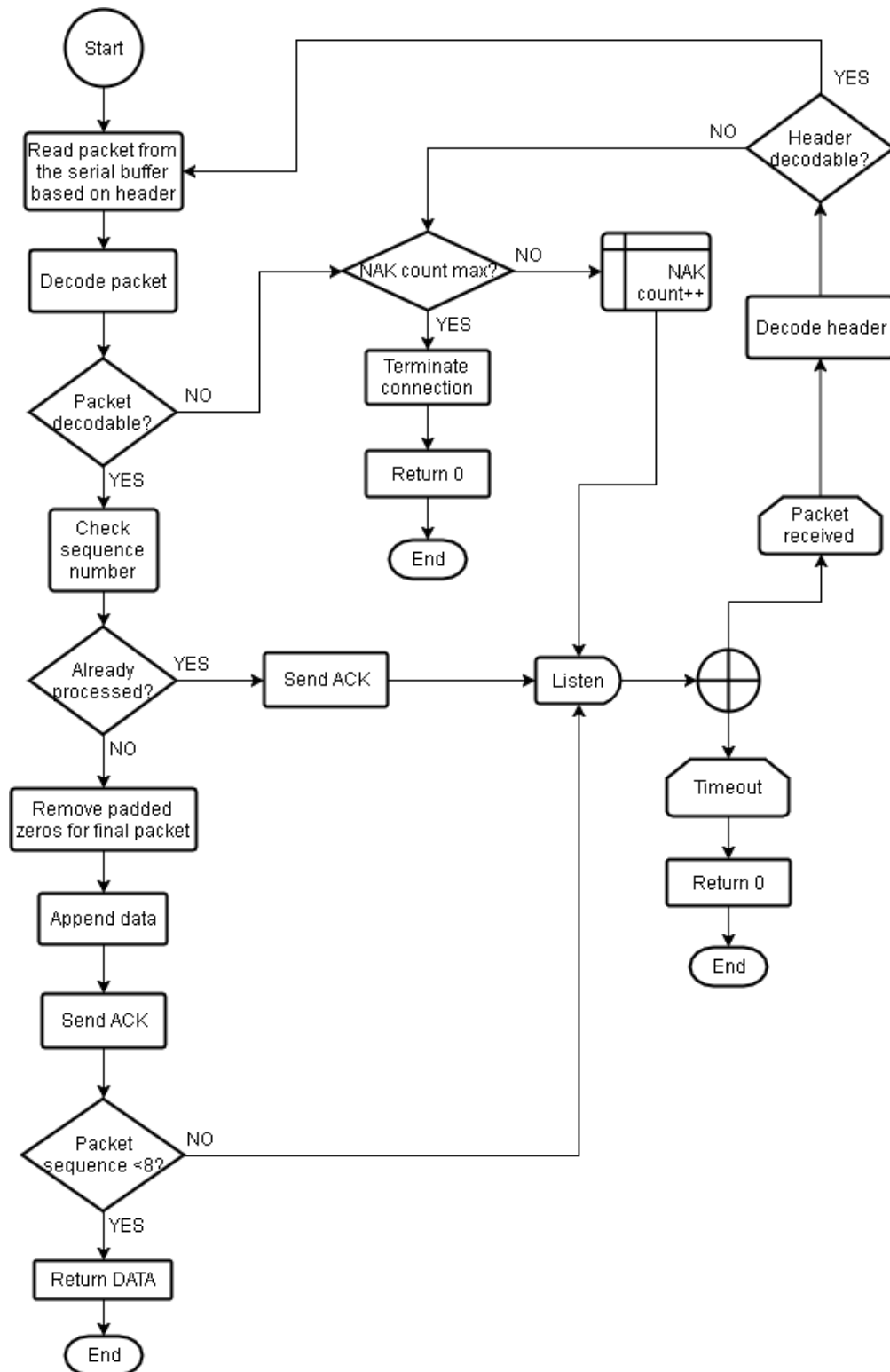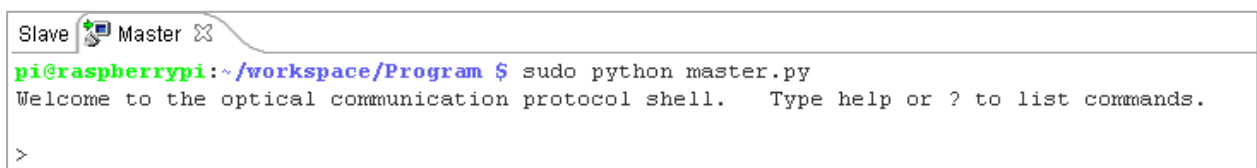
Figure 3.17: Get_data function – Master

3.5 Proof of concept

The implementation of the protocol went through the process of coding. As mentioned at the start of this chapter, the code for the master and slave ends are presented in the appendix. In this section, screenshots of the program's output are shown directly from the integrated development environment (IDE), where the protocol was developed. The program for the master and slave ends should be started the same way as any other Python scripts. For example, to run the program in the master side, one should navigate to the folder where the program is located through the command line, then execute the following command:

```
> sudo python master.py
```

With this, the following message should be seen, Figure 3.18.


```
Slave  Master ⋈
pi@raspberrypi:~/workspace/Program $ sudo python master.py
Welcome to the optical communication protocol shell.   Type help or ? to list commands.

>
```

Figure 3.18: Running the master program

After the program is started, it waits for commands to be inputted infinitely. At the slave side, although it logs and prints information at the output section, the program is not directly accessible, since it only responds to requests from the master.

The program contains the following commands:

> send [message]: sends a message to the slave end and displays it.
> getfile [file]: requests file from the slave end.
> location: updates the location of the of slave and prints it.

The error map file is also located in the same folder as the main program and presents the subsequent structure:

```
Relative height group, Relative angle group, Symbol error rate
```

As an example, considering the following entry:

```
30,20,0.00029
```

This line represents the relative height of the AUV, which is in the interval 30~40 cm, with the angle higher than 20º. This location has a SER of 0.00029 which is the rate to be returned.

3.5.1   First Output

As a first simulation, the text "This is a test message." was sent. The exchange between the master and slave ends is shown in Figure 19.

Figure 3.19: First outputs (Top output – master end; Bottom output – slave end)

To simplify the explanation, the image is divided in its relevant segments. It should be noted that whenever "\x" appears in the output, it represents a hexadecimal number with the following two characters.

Section A of Figure 3.19 signifies a request for the location. Having the packet design mentioned before in mind, the first line of this section is explained in Figure 3.20.
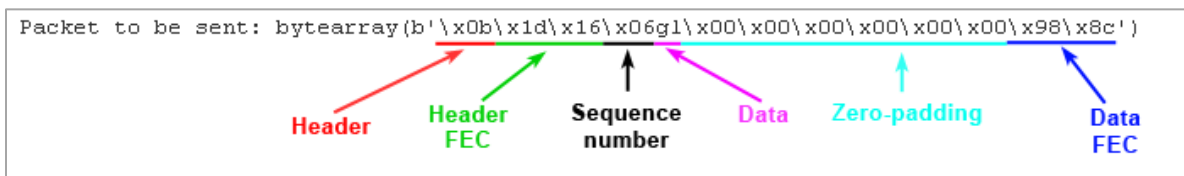
Figure 3.20: Code output – example of packet transmitted 1

The data is sent as "gl," which are the initials for "Get Location." As it can be seen, the sequence number is equal to the number of zeros used to pad the packet since the message is too short. It also sends the minimum 2 FEC symbols because the master still doesn't have the location

of the AUV. At the slave end, the first 3 bytes are received, being the header. After it decodes the header, it then reads the rest of the packet based on the header. After decoding the packet, the information is parsed, which the slave recognizes as being a request for data. The slave then sends the requested packets right away, as mentioned before (no ACK is sent here). This process turns the master into a receiver temporarily. The data received can be seen in this section as the location output. With the location, the SER can be now retrieved from the error map.

After receiving the location, the master now sends the command with the FEC bytes tuned, passing to section B of Figure 3.19. The message to be sent has the structure of Figure 3.21.
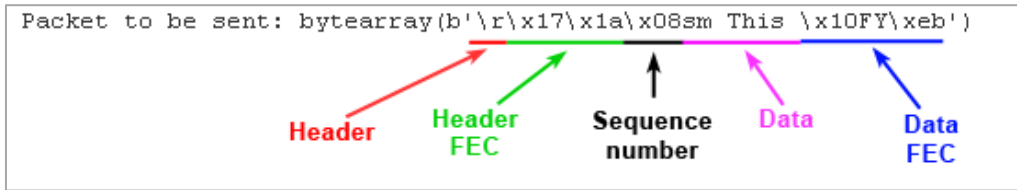
Figure 3.21: Code output – example of packet transmitted 2

As it can be seen, this packet contains the information "sm This ," where "sm" is the command to be recognized by the slave. The message is encoded with 4 FEC bytes this time due to the location. "This is a test message." is sent over 4 packets of 8. The ARQ response is displayed as in Figure 3.22.
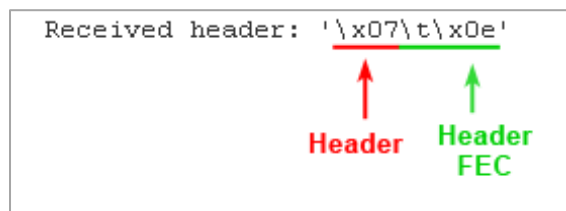
Figure 3.22: Code output – ARQ response

The figure follows the structure mentioned in the ARQ Packet design chapter. An ACK is represented as 0x07, while a NAK is 0x08. Since the message was successfully sent, it is displayed at the slave end's output, bottom Figure 3.19.

3.6 Simulation method

Simulations were conducted by following the requirements for this dissertation. Reliability and efficiency were used as simulation parameters.

3.6.1   Fixed Reed-Solomon coding rates

The reliability of the RS algorithm was tested using predefined code rates, which could then be used as comparison to the theoretical values and the proposed solution. The probability of having a packet error while decoding can be seen in the channel modeling, equation (3). This can be used to get the probability of decoding a packet, given a symbol error probability P_s

$$P_{dec} = 1 - \sum_{k=\frac{N-K}{2}+1}^{N} \binom{N}{k} P_s^k (1 - P_s)^{N-k}, \qquad (9)$$

Since RS corrects data on a byte level, tests were directed on a symbol-basis, rather than in bits. The simulation platform for this part is shown in the appendix.

3.6.2 System simulations

Tests involved editing the master end's program for the proposed solution. This way, simulations could be performed to test the slave end's (AUV) parameters. Similar to the program written above, the following vectors were created:

```
1.  # Number of decoded packets
2.  dec=[0] * lpe
3.  # Number of incorrect packets
4.  idec=[0] * lpe
5.  # Number of decoded headers
6.  hdec=[0] * lpe
7.  # Number of retransmissions
8.  naks=[0] * lpe
9.  # Number of uncoded packets without error
10. ndec=[0] * lpe
11. # Total number of symbols received
12. nsym=[0] * lpe
13. # Array of data lengths received (used to calculate efficiency)
14. veff=[0] * lpe
```

Where "lpe" is the length of the probabilities' array to be tested. Analogous to the RS testing algorithm, these arrays were populated by introducing them on key points of the program. For example, for the case of decoded packet count, the following edit was made:

```
1.  ...
2.  # Try decoding
3.  rs_packet=packet + ''.join(b'0' for x in range(18-nk))
4.  p = reedsolo.rs_correct_msg(rs_packet, 18, 0, 2, [i for i in range(header,(dlengt
    h+18))])[0]
5.  # Compare to original packet
6.  if p == original:
7.      dec[i]+=1
8.  ...
```

The channel's symbol error rate was recreated by using the random() function from Python just before the packet is sent, implemented the following way:

```
1.  ...
2.  # Channel simulation
3.  for x,y in enumerate(en):
4.      # Add errors based on probability to be tested
5.      if random.random() < pe[i]:
6.          # Avoid writing a symbol that is the same as the original
7.          if en[x]!=1:
8.              en[x]=1
9.          else:
10.             en[x]=0
11.
12. send_loop_tout(en)
13. ...
```

In this case, "en" is a packet that has been encoded with the error correction algorithm.

Finally, after the arrays are populated, the plots were made using the same method from the RS' simulation case.

Reliability

Probabilities P_s were sampled in the interval $]0,1[$. Since each iteration of P_s requires *n* packets to be sent, processing could've been an issue if too many points were tested, so only the following probabilities were tested:

```
# Probability of errors
pe = [0.001, 0.01, 0.03, 0.05, 0.1, 0.15, 0.25, 0.40, 0.5]
```

Efficiency

Efficiency was calculated based on the following formula

$$\text{H} = \frac{Data \times Number\ of\ decoded\ packets}{Packet * Total\ number\ of\ packets + Retransmissions} \times 100, \quad (10)$$

where H is the efficiency in percent and "Data" is the number of symbols that are part of the message.

Error-correction control simulations

Simulations conducted above present results according to each probability given. However, as described before, the AUV does not possess perfect localization techniques. The error-correction control system accounts for the failure of these systems, monitoring the transmission at all times to control the quality of the communication. The reliability of this feature was tested through two case-scenarios where the probability that is passed to the FEC control is different from the one used to simulate the channel's errors, which can be seen in the code above.

3.7 Conclusion

In this chapter, a system engineering approach was used to develop the optical communication protocol described. The process followed the key features presented below:

- – Analyzation of the requirements for this project.
- – Design of the system architecture.
- – Implementation of the protocol.
- – Simulation and verification of the results.

The results and respective discussion of the simulations are revealed in the next chapter.

Chapter 4 - Results and discussion

In the present chapter, the results from the simulations according to the methodology described before are shown. The measurements presented below test the Reed-Solomon algorithm and the proposed solution. These tests were made with 10000 packets sent in all cases except for the error-correction control system.

4.1 Fixed Reed-Solomon coding rates

As mentioned in the requirements, fixed Reed-Solomon code rates were the first tests to be performed, with the objective of being used for comparison against variable FEC coding. Three code-rates were tested, $^9/_{11} \cong 0.8$, $^9/_{17} \cong 0.5$ and $^1/_3 \cong 0.3$, which is equivalent to decoding rates of $^1/_9, ^4/_9$ and $^9/_9$. The simulations are respectively shown in Figures 4.1, 4.3, 4.4, with the percentage of decoded packets as a function of symbol error probability, which is a theoretical equivalent to symbol error rate. Figure 4.2 presents the number of undetected errors as a function of symbol error probability, scaled to 20%. In the drawn plots, "Uncoded" is the simulation using no FEC coding, "Theory" is the result of the formula present in the simulation method for a given probability, and "Experimental" shows the result of the executed algorithm.
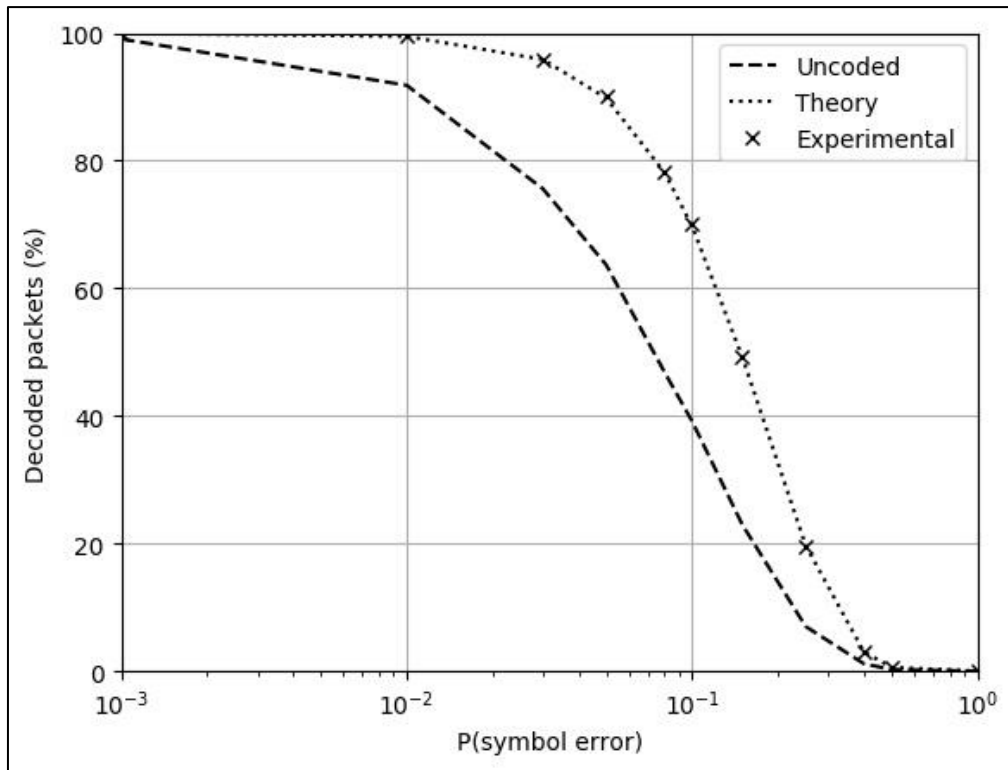
Figure 4.1: Fixed code-rate decoding capability – 9/11 (10000 packets simulated)
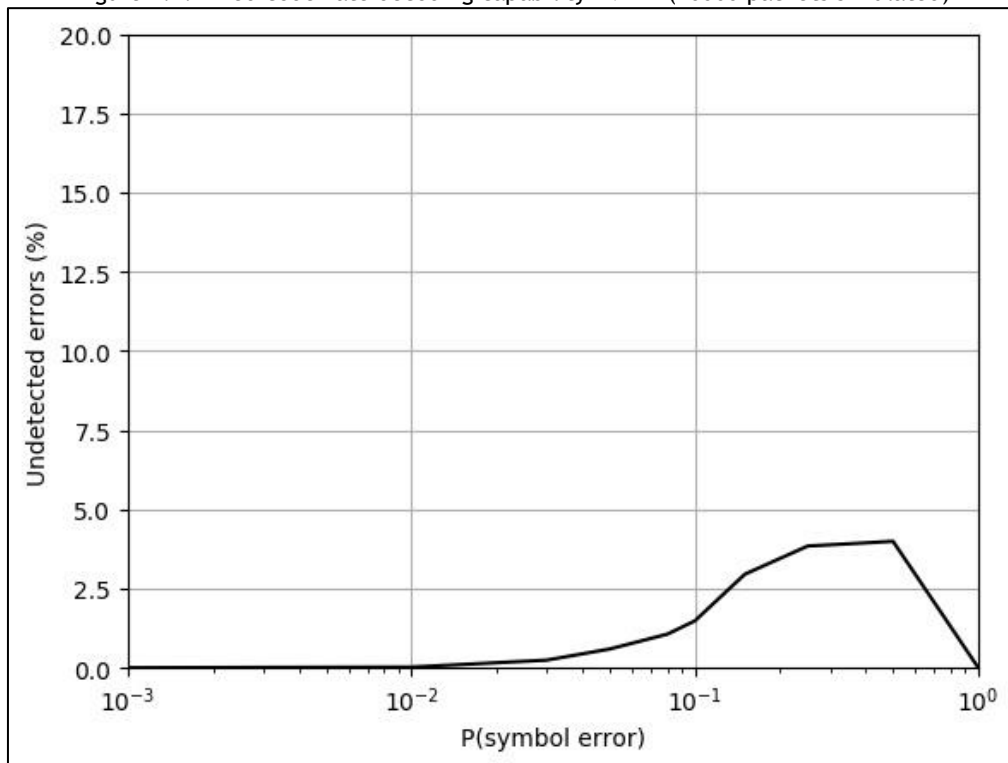
Figure 4.2: Undetected errors during decoding – 9/11 (10000 packets simulated) – Scaled to 20%
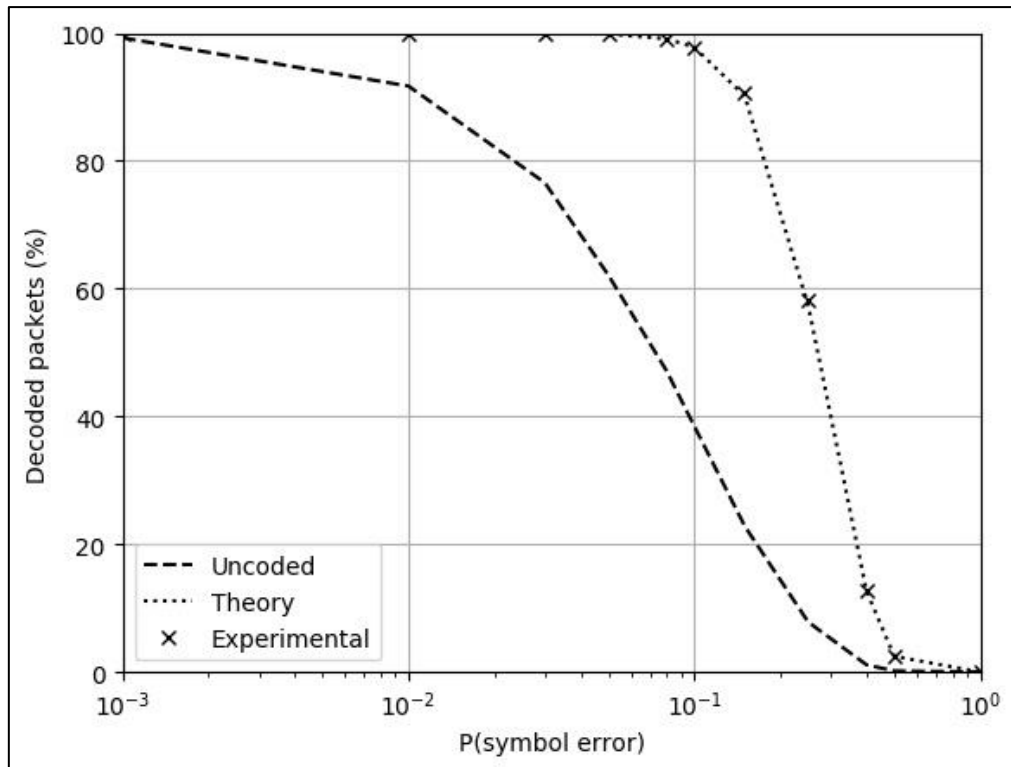
Figure 4.3: Fixed code-rate decoding capability – 9/17 (10000 packets simulated)
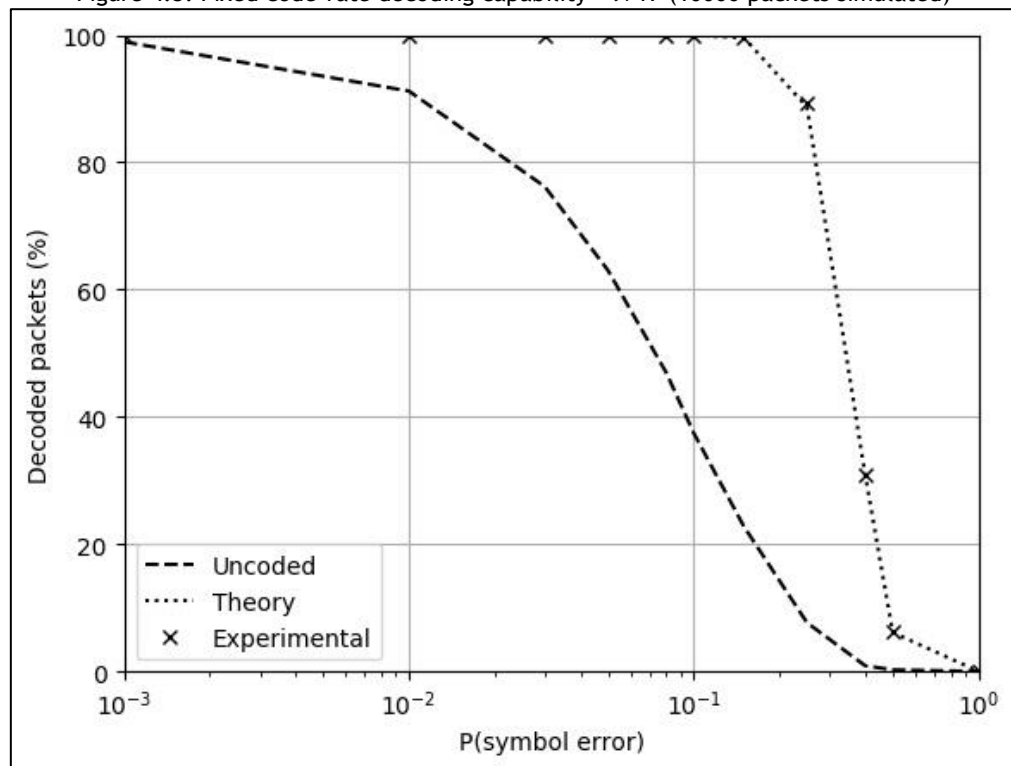
Figure 4.4: Fixed code-rate decoding capability – 1/3 (10000 packets simulated)

The results are as expected. All the experimental values coincide with the theoretical values, which proves that the number of packets used to test is acceptable. Among these three simulations, only the first case (code rate 9/11) shows undetected errors, Figure 4.2. This is explained by the fact that the code rate 9/11 uses 2 FEC bytes. As stated by the singleton bound, the maximum

number of errors that the algorithm can detect is $n - k + 1 = 11 - 9 + 1 = 3$. High symbol error probabilities can easily overcome this bound in a 11-packet size, which results in higher undetected errors.

4.2 Proposed system

The proposed system's simulations were led to test if the requirements were met while emulating the program in a practical situation. The simulations below test two case-scenarios: a situation where only small packets are received and another where variable packet lengths are received. Besides case-scenarios, tests also include: header and ARQ, and error-correction control.

4.2.1 Case-scenario 1: small packets

This scenario will emulate a situation where only commands are received, which are usually lower than 8 bytes. It can be interpreted as a common situation from the master's side, which will mostly be used to send short messages and requests. Two plots can be seen below: Figure 4.5 shows the percentage of decoded packets and Figure 4.6 displays the efficiency of the proposed solution for small packets.
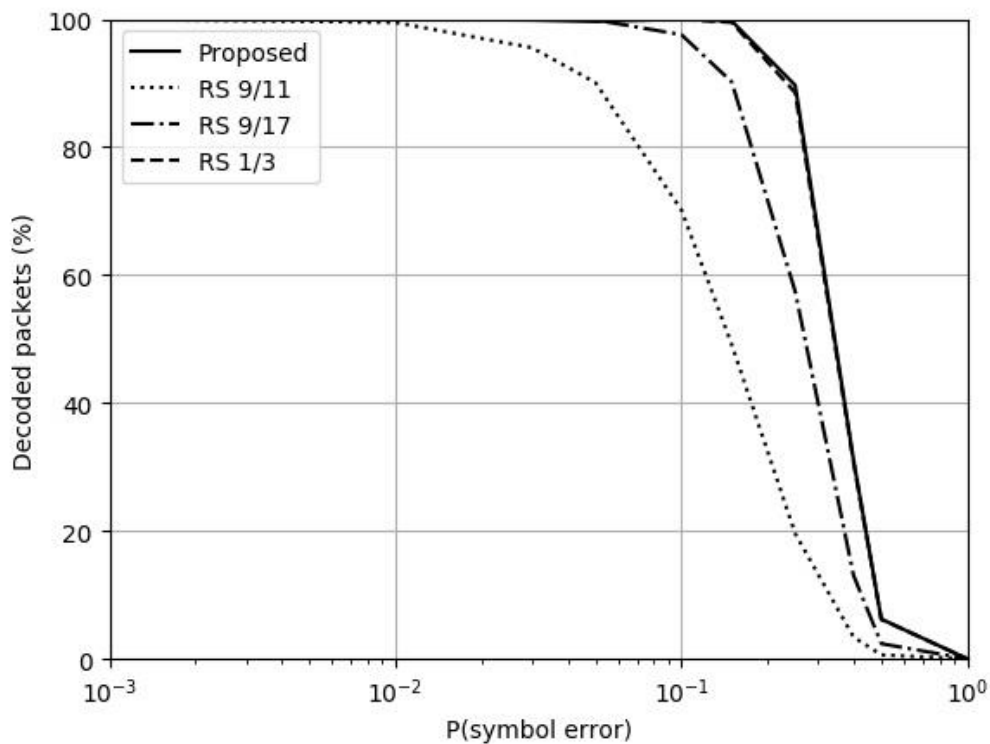
Figure 4.5: Proposed system's decoding capability – case scenario 1 (10000 packets simulated)
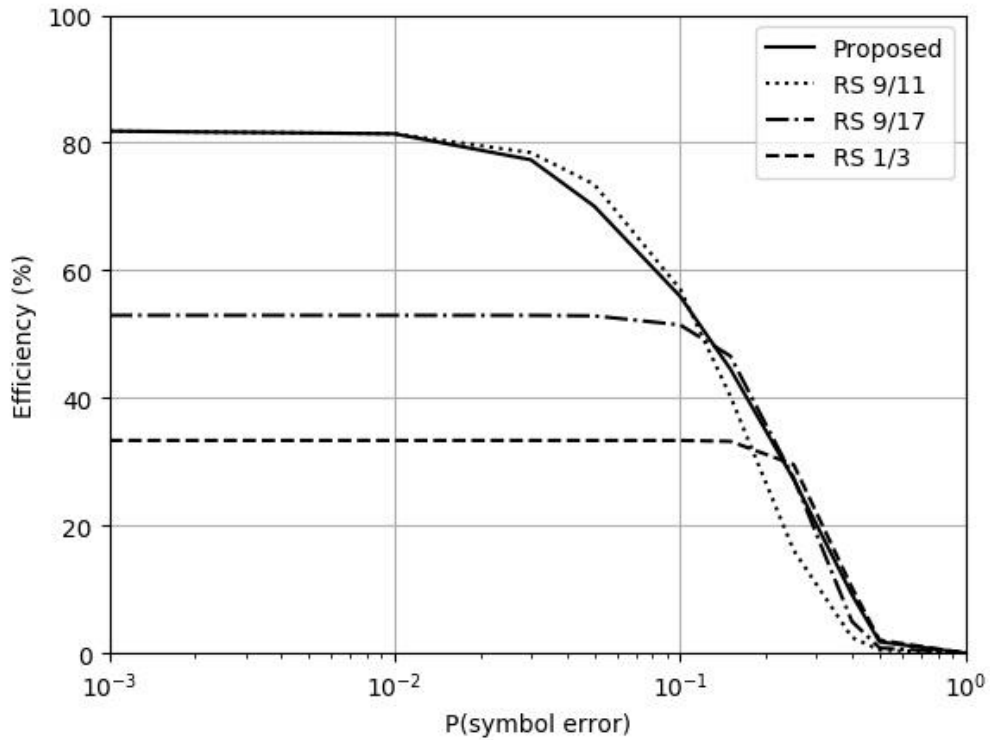
Figure 4.6: Efficiency plot - case-scenario 1 (10000 packets simulated)

The results observed are as expected, showing that despite being irregular, not only does the proposed solution possess the same decoding capability as the fixed RS' lowest code rate, but it also has an efficiency close to the highest fixed RS code rate. The irregular behavior of the plot can be explained by the fact that the N-K values must be multiples of 2. Not only that, the line of the efficiency plot is slightly lower than the code rate 9/11 because maximum reliability must be ensured first.

4.2.2 Case-scenario 2: variable-length packets

As an addition to the case-scenario above, now variable packet sizes are allowed. This can be seen as the slave side, which will most likely be the one sending large data such as images and video. In the same manner, Figure 4.7 shows the percentage of decoded packets and Figure 4.8 displays the efficiency of the proposed solution for variable length packets.
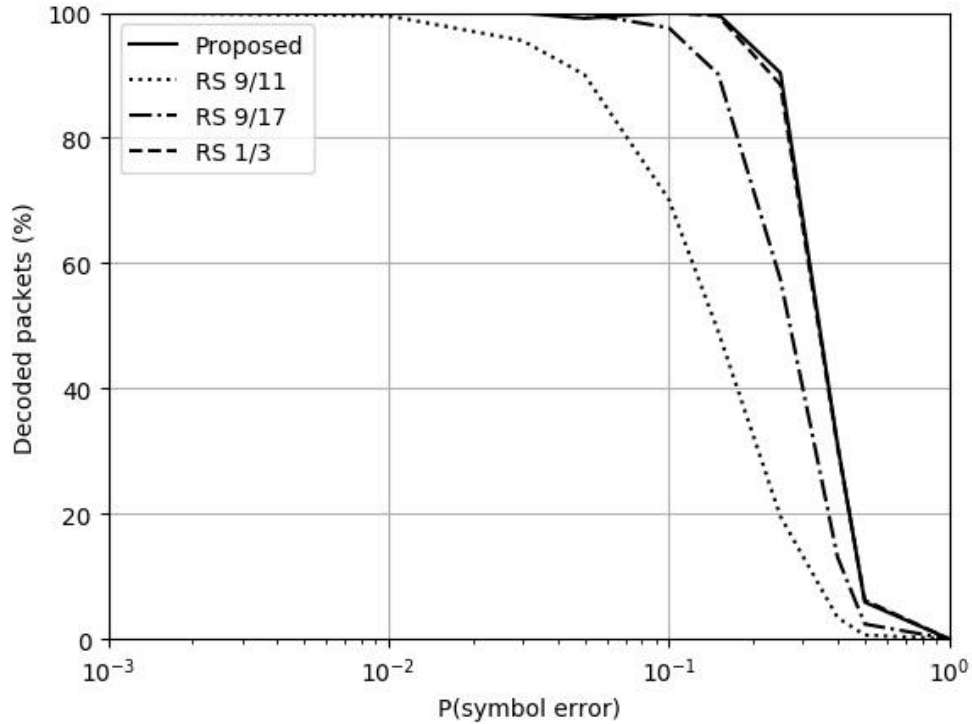
Figure 4.7: Proposed system's decoding capability - case-scenario 2 (10000 packets simulated)
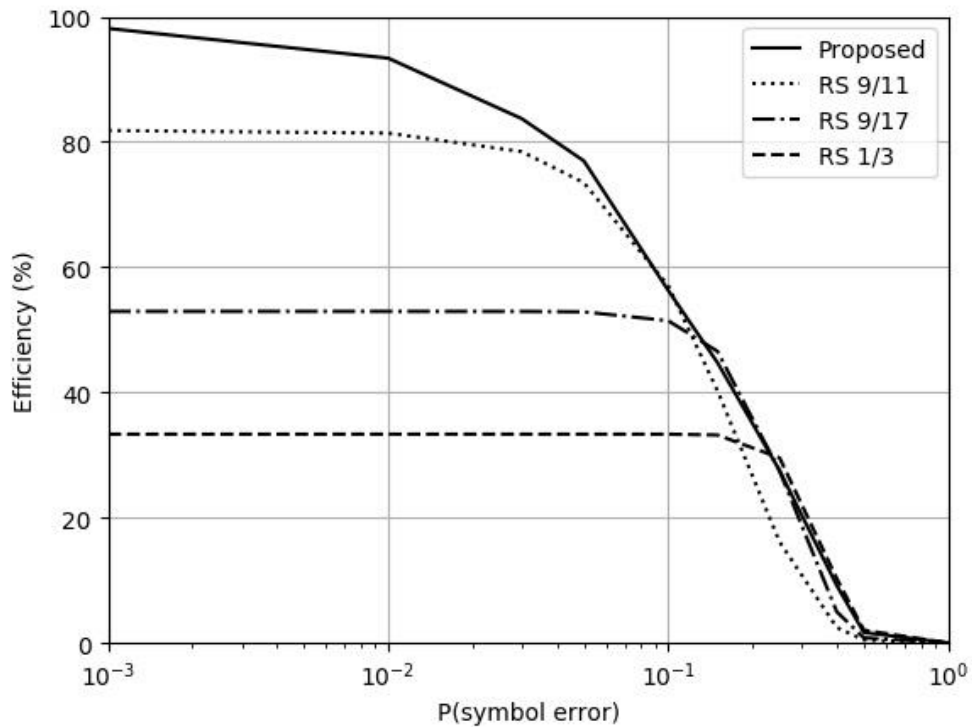
Figure 4.8: Efficiency plot - case-scenario 2 (10000 packets simulated)

The results show that using variable packet lengths increases efficiency of the system, which was to be expected, since packets of higher dimensions feature lower ratio of overhead. However, the equation (3) from channel modeling needs to be interpolated to limit the packet sizes. If more iterations were made, better approximations could be observed, further increasing the efficiency of the proposed solution.

4.2.3   Header and ARQ decoding

Figure 4.9 shows the decoding capability of the header and the ARQ system, as they both possess the same design. The coding rate of the header is $1/3$, which represents a decoding rate of 1.
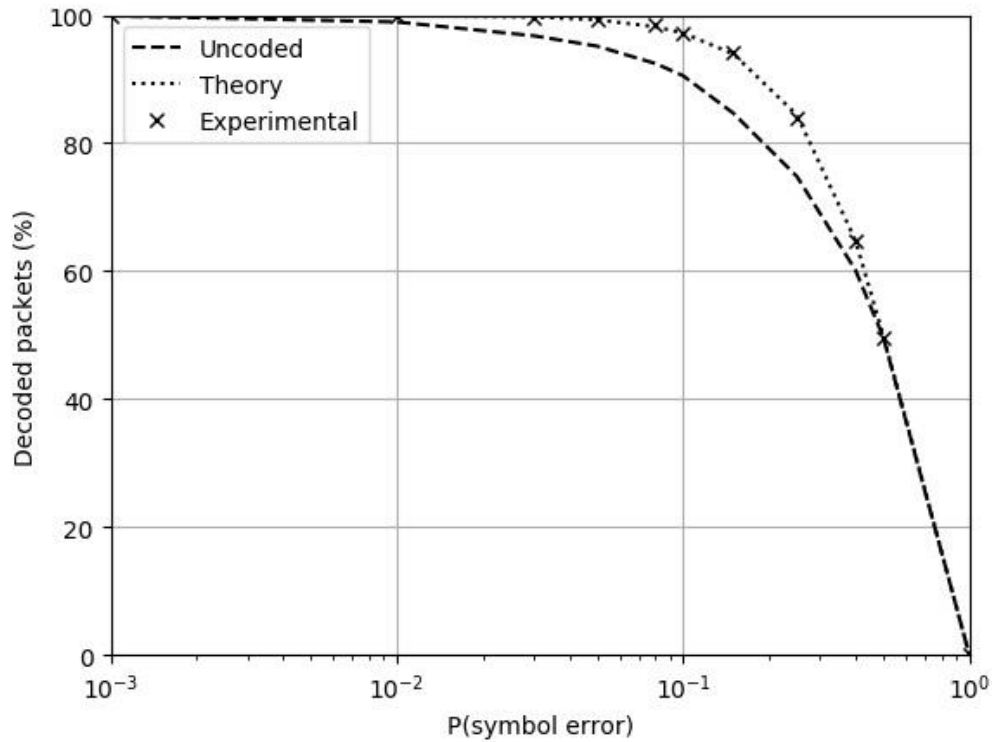
Figure 4.9: Header and ARQ decoding capability (10000 packets simulated)

Results show that the design of the header can be successfully attached to the packets without hindering efficiency, as its decoding capability outperforms the system's decoding ability in both cases tested above. However, the header in the proposed system is not adequate for multi-byte errors. This is because, being 3 bytes long, it can only decode one error in any of those three bytes.

4.2.4 Error-correction control system

Figure 4.10 and Figure 4.11 test the capability of the FEC control mechanism through two case-scenarios where 2000 packets were sent in total. For the first simulation, the parameters used were: $P_s^A = 0.08$, which is the apparent error probability that is fed to the error-correction control, and $P_s^R = 0.1$, the real error probability. These two probabilities correspond to code rates of 9/11 and 9/13 respectively. Based on the Table 3.2 from the error-correction systems, the chosen parameters were: $n = 1000$ and $tolerance = 33$. The plot below displays the N-K values from the from the code rates N/K as a function of the number of packets sent in a transmission.
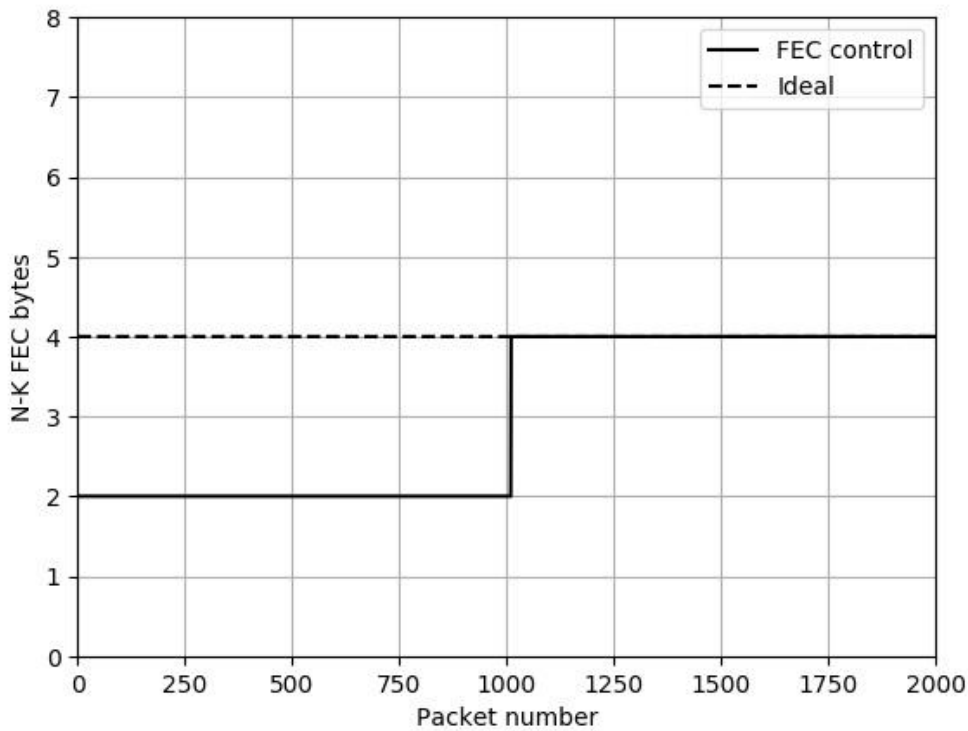
Figure 4.10: Evolution of N-K with FEC control (2000 packets simulated) – P^A=0.08, P^R=0.1

The intended value on the plot was calculated using the equation (9) from the variable error-correction sub-chapter for P_s^R.

For the second case-scenario, the proposed error-correction system was simulated with the probabilities reversed, Figure 4.11. Here, $P_s^A = 0.1$ and $P_s^R = 0.08$.
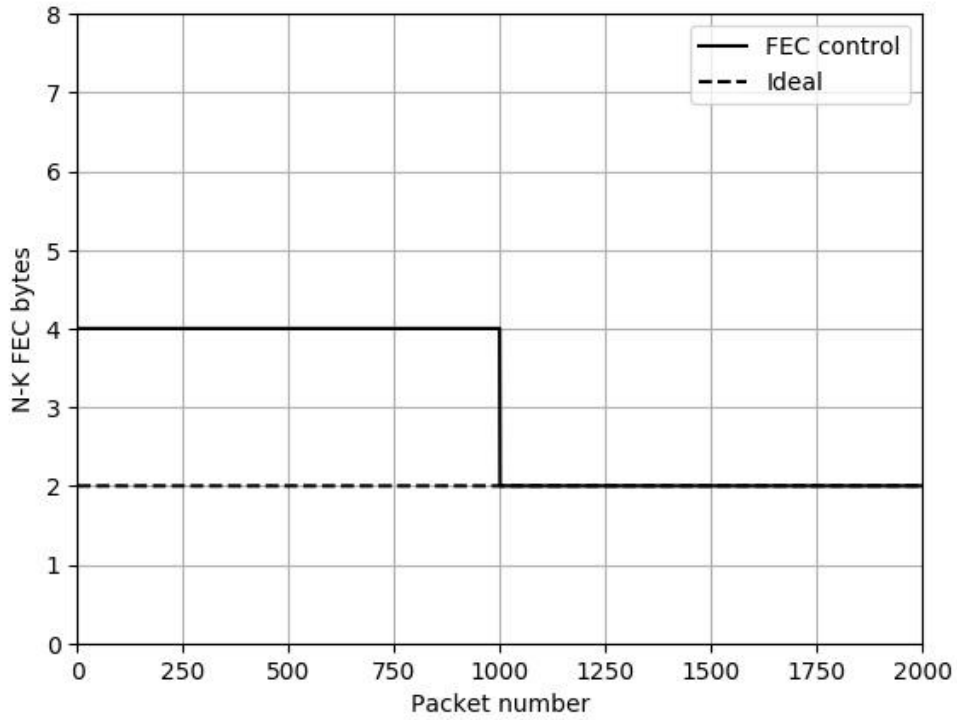
Figure 4.11: Evolution of N-K with FEC control (2000 packets simulated) – P^A=0.1, P^R=0.08

As it can be seen, the FEC control is robust against disparities from the probabilities given by the AUV and the real one, as both test-cases converged to the intended value. However, this error-correction control system can only correct location differences when it doesn't represent a difference of $|(N-K)_A - (N-K)_R| > 2$. This is because after the correction, there isn't a probability for the FEC control to base on.

Chapter 5 - Conclusions

The proposed solution proves to be as reliable as the lowest Reed-Solomon's code rate, which presents a decoding rate of 1. Still, it is observed a maximum increase of 45% in efficiency in comparison to the same code rate. This proves that the requirements have been met, as well as the objectives proposed at the beginning of this dissertation. The design of the header and ARQ packets also proves to be dependable, with a maximum of 37% increase in packet decoding capability against the tested case-scenarios. The implementation of the error-correction control also proves to be essential, as the AUV's localization system should not be entirely relied upon. Results have shown that the system is able to correct SERs related to a difference of 2 encoding symbols. It should be noted that although the design of the protocol was done for optical devices of lower cost, implementation of this system is independent of the devices, given that fine-tuning is made beforehand.

5.1 Completed tasks

The objectives proposed for this dissertation were completed through several tasks:
- Analysis of the system, with the observation of different behaviors for the near and far field of the optical beams.
- Research of the different algorithms used in recent projects for error-control, in both land and underwater environments.
- Investigation of the various aspects to be implemented in the protocol, such as the main system architecture, design of the packets, channel modeling, error-correction systems, automatic repeat requests, and the design of the code. Here, an enhancement to the adaptive error-correction mechanism was proposed.
- Implementation of the program, coded in two separate ends: master and slave.
- Coding of the simulation platform, with an analysis of what should be tested.
- Simulation of the protocol in diverse situations.

5.2 Contributions

Major contributions from this dissertation include:
- Creation of the error-correction control system, able to handle the uncertainty of the AUV localization.
- Implementation of a communication protocol using hybrid ARQ type II with incremental redundancy in Python that handles both near and far-field behaviors.

Aside from the major contributions, the following sub-contributions were made:
- Creation of a testing platform to calculate the experimental errors for a given number of packets.
- Creation of a simulation platform for Reed-Solomon algorithms.
- Implementation of an adaptive error-correcting mechanism based on the BSC model.
- Design of a robust 1-byte header long for variable packet lengths in a communication line type of PPP.

5.3 Future work

Although the goals for this dissertation have been achieved in its entirety, it is still possible to further improve the work done. Below is a list of improvements that could build upon the results in order to enhance the capabilities of the protocol.

- Using either the sequence byte or one more byte to encode location information from the AUV in the packet might drastically improve the communication line, once the master end would be able to receive location updates close to real-time.

- Editing the header to implement the functionality of the sequence number, reducing the constant overhead. This is possible by using the integers between the N-K symbols, which need to be multiples of 2.

- Increasing the capability of the error-correction control by simulating a probability after a correction, which could potentially correct higher probability discrepancies.

- Adding the application layer, where a hierarchy-based system would select the code rate based on how important a message is, while adding more features such as file compression with the purpose of sending over noisy channels.

- Optimizing the speed of the protocol by using a more complex language such as C and implementing faster functions with an enhanced design, able to process data for high-speed communications.

Aside from these improvements, using the program on a real-case scenario to fine-tune the protocol would ensure a quality communication line for optical devices.

Appendix

In this chapter, the code developed to implement the optical communication protocol is presented.[2] It is divided in four parts:

- Experimental error test (test.py), comprehending the code to test the experimental values required for the proposed error-correction control.
- Master (master.py), implementing the main functions required to function in the station's end.
- Slave (slave.py) – Changed functions, implementing the main functions required to function in the AUV's end. Only the changed functions are displayed from master, as most of the program is identical.
- Reed-Solomon simulation (sim_rs.py), containing the main simulation platform for the Reed-Solomon algorithm.

---

[2] A full-version of the code can be seen at https://github.com/helderjbe1/ocp.

Experimental error test (test.py)

```
1.  import reedsolo
2.  import random
3.
4.  # Factorial of a number
5.  def factorial(n):return reduce(lambda x,y:x*y,[1]+range(1,n+1))
6.
7.  # Calculation of packet decoding error probability
8.  def PDEP(p, N, nk):
9.      min = (nk/2)+1
10.     pdep=0
11.     for k in range(min, N+1):
12.         pdep += (float(factorial(N))/(factorial(k)*factorial(N-k)))*(p**k)*((1-p)**(N-k))
13.
14.     return pdep
15.
16. # Init reedsolomon tables
17. reedsolo.init_tables(0x11d)
18.
19. # Probability of error
20. pe = 0.08
21.
22. # Number of trials
23. n= 100
24.
25. # Number of packets to test
26. narr= 1
27.
28. # Packet to be tested
29. nk = 2
30. packet=reedsolo.rs_encode_msg('123456789', nk)
31. l=len(packet)
32.
33. # Decoding number array
34. dec=[0] * n
35.
36. # Get theoretical values
37. # Probabilities of decoding
38. pt = 1-PDEP(pe, l, nk)
39. # Multiply by packets
40. dect= pt*narr
41.
42. for num in range(n):
43.     # Simulation loop
44.     for i in range(narr):
45.         en = packet[:]
46.         # Channel simulation
47.         for x,y in enumerate(en):
48.             # Add errors based on probability to be tested
49.             if random.random() < pe:
50.                 # Avoid writing a symbol that is the same as the original
51.                 if en[x]!=1:
52.                     en[x]=1
53.                 else:
54.                     en[x]=0
55.
56.         # Test if decodable
57.         try:
58.             de = reedsolo.rs_correct_msg(en, nk)[0]
59.             # Correct decoding?
60.             if de == packet[:-nk]:
61.                 dec[num]+=1
62.         except reedsolo.ReedSolomonError:
```

```
63.                continue
64.
65. # dec - Decoded number
66. # dect - Theoretical value
67. print max(dec)-dect
68. print min(dec)-dect
```

Master (master.py)

```
1.  import serial
2.  import logging
3.  import cmd
4.  import csv
5.  import cStringIO
6.  import reedsolo
7.  import time
8.  import os
9.  import math
10.
11. ################ SETUP
12.
13. # Logging init
14. logging.basicConfig(
15.     filename='master.log',
16.     format='%(asctime)s %(message)s',
17.     datefmt='%m/%d/%Y %I:%M:%S %p',
18.     level=logging.DEBUG
19.     )
20.
21. # Set up serial port
22. ser = serial.Serial(
23.     port='/dev/ttyAMA0',
24.     baudrate=921600,
25.     parity=serial.PARITY_NONE,
26.     stopbits=serial.STOPBITS_ONE,
27.     bytesize=serial.EIGHTBITS,
28.     #inter_byte_timeout=0.1,
29.     timeout=0
30. )
31.
32. # Init precomputed tables for Reed Solomon
33. reedsolo.init_tables(0x11d)
34.
35. # Pre-generation of polynomials for faster encoding
36. gen=reedsolo.rs_generator_poly_all(19)
37.
38. ################ DEFINES
39.
40. # Protocol bytes
41. ACK = b'\x07'
42. NAK = b'\x08'
43. ack_pack = reedsolo.rs_encode_msg(ACK, 2, 0, 2, gen[2])
44. nak_pack = reedsolo.rs_encode_msg(NAK, 2, 0, 2, gen[2])
45. #For send mode a cancel with packet length must be issued to be recog-
    nized by the receiver
46.
47. # Relative location of the AUV. Init with None
48. # Height, angle
49. loc = [None, None]
50.
51. # Error map file (CSV)
52. mapFile='error_map'
53.
54. # Timout management
55. TOUT_recv=0.7 # Timeout seconds for ARQ system
56. TOUT_send=0.5 # Timeout seconds for send function
57.
58. # Limits of the packets (SER)
59. lim236 = 0.0179
60. lim64 = 0.0596
61.
62. # Updater vars
```

```
63. updater_n = 200
64. updater_tol = 6
65. # Init circular buffer
66. updater_buf = [-1 for init1 in range(updater_n)]
67. updater_prob = 0
68. updater_count = 0
69. updater_key = 1
70. updater_lastloc = loc
71.
72. ############################################
73.
74. # Shutdown properly
75. def shutdown():
76.     logging.debug('Shutdown request')
77.     logging.shutdown()
78.     ser.flushOutput()
79.     ser.flushInput()
80.     ser.close()
81.     exit()
82.
83. # Init function
84. def init():
85.     # Serial port initialization
86.     if ser.isOpen():
87.         try:
88.             ser.flushInput() #flush input buffer
89.             ser.flushOutput()#flush output buffer
90.         except Exception, ef:
91.             print ("Error flushing buffers")
92.             logging.error("Error flushing buffers")
93.             return 0
94.     else:
95.         print ("Error opening serial port")
96.         logging.error("Error opening serial port")
97.         return 0
98.
99.     return 1
100.
101.         # Factorial of a number
102.         def factorial(n):return reduce(lambda x,y:x*y,[1]+range(1,n+1))
103.
104.         # Calculation of packet decoding error probability (binomial form)
105.         def PDEP(p, N, nk):
106.             min = (nk/2)+1
107.             pdep=0
108.             for k in range(min, N+1):
109.                 # Calculate PDP with the formula from literature
110.                 pdep += (float(factorial(N))/(factorial(k)*factorial(N-
    k)))*(p**k)*((1-p)**(N-k))
111.
112.             return pdep
113.
114.         # Cumulative distribution function
115.         def CDF(x):
116.             return (1.0 + math.erf(x / math.sqrt(2.0))) / 2.0
117.
118.         # Approximation of binomial cdf with continuity correction
119.         def NPDEP(p, N, nk):
120.             min = (nk/2)+1
121.             return 1-CDF((min-0.5-(N*p))/math.sqrt(N*p*(1-p)))
122.
123.         # Iteration of the best N-K values
124.         def NK(p, K):
125.             nkarr= [0] * 9
126.             for i in range(9):
127.                 nk = (i+1)*2
```

```
128.              if K >= 100:
129.                  nkarr[i] += (float(K)/(nk+K)) * (1-NPDEP(p, (K+nk), nk))
130.              else:
131.                  nkarr[i] += (float(K)/(nk+K)) * (1-PDEP(p, (K+nk), nk))
132.
133.          nk = (nkarr.index(max(nkarr))+1)*2
134.
135.          # Limit packets of 237 to 4 bytes to avoid singleton bound errors
136.          return nk+2 if nk<=2 and K>=100 else nk
137.
138.      # Interpolate the locations given by the AUV so that it's possi-
     ble to group the ByER (byte error rate)
139.      # for the angle, it's divided in two parts: from 0~20 and >20
140.      # for the height, it is grouped in steps of 10cm.
141.      # all of the values on the error map might be changed with ease in the fu-
     ture.
142.      def interpolate():
143.          if loc == [None, None]:
144.              return [None, None]
145.          else:
146.              grouph = int(math.floor(float(loc[0])/10)*10)
147.              groupa = 0 if loc[1]<20 else 20
148.              return [grouph, groupa]
149.
150.      # Search inside the csv file for the location
151.      # Error map file structure:
152.      # [Height group], [Angle group], [Initial error rate]
153.      def get_fec():
154.          l = interpolate()
155.          if l == [None, None]:
156.              return 0
157.
158.          reader = csv.reader(open(mapFile+'.csv', 'rb'), delimiter=',')
159.          for row in reader:
160.              if l[0]==int(row[0]) and l[1]==int(row[1]): #col 1 and 2
161.                  return float(row[2])
162.
163.          return 0
164.
165.      # Updater ( Error-correction control )
166.      # Checks last updater_n bytes to see if there is a discre-
     pancy between the amount of NAKs and ACKs, based on the probability of packet er-
     ror
167.      # type: 0 if NAK, 1 if ACK
168.      def updater(type, p=0):
169.          global updater_buf
170.          global updater_count
171.          global updater_key
172.          global updater_prob
173.          global updater_lastloc
174.
175.          # Check if location changed first and reset if it has
176.          if loc != updater_lastloc:
177.              updater_key = 1
178.              updater_count = 0
179.              updater_buf = [-1 for new in range(updater_n)]
180.              updater_lastloc = loc
181.
182.          if updater_key and p>0:
183.              if p < lim236:
184.                  nk = NK(p, 237)
185.                  updater_prob = NPDEP(p, 237+nk, nk)
186.              elif p < lim64:
187.                  nk = NK(p, 65)
188.                  updater_prob = PDEP(p, 65+nk, nk)
189.              else:
```

```
190.                           nk = NK(p, 9)
191.                           updater_prob = PDEP(p, 9+nk, nk)
192.                    updater_key = 0
193.
194.              # Update buffer
195.              if updater_count == 0:
196.                    updater_buf[updater_buf.index(-1)] = type
197.
198.                    # Check if buffer full
199.                    if -1 not in updater_buf:
200.                           cnt = updater_buf.count(0)
201.                           if cnt > updater_prob * updater_n + round(updater_tol/2.0):
202.                                  updater_count += 2
203.                           elif cnt < updater_prob * updater_n - round(upda-
     ter_tol/2.0) :
204.                                  updater_count -= 2
205.
206.        # Non-blocking header read with variable timeouts.
207.        # tout: timeout value
208.        # returns: header decoded, error, or None
209.        def hread(tout):
210.              start = time.time()
211.              while tout<=0 or (time.time() - start) < tout:
212.                    if ser.inWaiting() >= 3:
213.                           try:
214.                                  #sim
215.                                  r = ser.read(3)
216.                                  if ord(r[0])>9:
217.                                         print("Received header: %s\n" % repr(r))
218.                                  else:
219.                                         print("Received response: %s\n" % repr(r))
220.                                  header = reedsolo.rs_correct_msg(r, 2)[0]
221.
222.                                  return ord(header)
223.                           except reedsolo.ReedSolomonError as e:
224.                                  return 'Error'
225.
226.              ser.flushInput
227.              return None
228.
229.        # Non-blocking header read with variable timeouts.
230.        # val: amount of bytes to read
231.        # returns: packet read, error or None
232.        def pread(val):
233.              # If it's a retransmission, pass the value to 2
234.              if val == 3:
235.                    val = 2
236.              start = time.time()
237.              while time.time() - start < TOUT_recv:
238.                    if ser.inWaiting() >= val:
239.                           return ser.read(val)
240.
241.              ser.flushInput
242.              return None
243.
244.        # Terminator of a communication exchange
245.        # Reads until timeout and if something is read then send a response accor-
     dingly
246.        # mode: 1- ACK, 0 - NAK
247.        def terminator(mode):
248.              while True:
249.                    r = hread(TOUT_recv)
250.                    if r!=0 and r!=None:
251.                           if mode==1: #ACK
252.                                  ser.write(ack_pack)
253.                           else:
```

```
254.                         ser.write(nak_pack)
255.                 else:
256.                     return
257.
258.         # Get data when a request is sent
259.         def get_data(header, file=None):
260.             seq_last=int(100) # Random number different from 1
261.             buffer=[]
262.             retr_seq = 0 # Retransmission sequence
263.
264.             while True:
265.                 # Test if there was no error in request sent
266.                 if header==None:
267.                     print("In function get_data(): Error receiving what was re-
    quested")
268.                     logging.critical("In function get_data(): Error recei-
    ving what was requested")
269.                     return None
270.                 elif header=='Error':
271.                     ser.flushInput()
272.                     ser.write(nak_pack)
273.                     header=hread(TOUT_recv)
274.                     continue
275.                 elif 6 < header < 9:
276.                     # It is an ACK or a NAK
277.                     print("In function get_data: Expected data, ARQ res-
    ponse found")
278.                     logging.critical("In function get_data: Expec-
    ted data, ARQ response found")
279.                     return None
280.                 else:
281.                     break
282.
283.             while True:
284.                 # Check if it's a retransmission
285.                 temp = pread(header)
286.                 if temp==None:
287.                     print("In function get_data: Expected packet")
288.                     logging.critical("In function get_data: Expected packet")
289.                     return None
290.
291.                 # Check if it's a retransmission
292.                 if 2 <= header <= 3:
293.                     # It's a retransmission
294.                     if retr_seq != header: # Append the FEC sym-
    bols only if the sequence is different from the last one
295.                         packet += temp
296.                         header = len(packet)
297.                 else:
298.                     packet = temp
299.
300.                 # Calculate N-K based on packet length
301.                 if header >= 237: # 236+1 packet
302.                     dlength = 237
303.                 elif 65 <= header < 237: # 64+1 packet
304.                     dlength = 65
305.                 else: # 8+1 packet
306.                     dlength = 9
307.
308.                 nk = header-dlength
309.
310.                 # header = data + current FEC
311.                 # dlength = data
312.                 # nk = FEC
313.                 # dlength + 18 = max packet
314.                 try:
```

```
315.                    # Parse sequence number and data
316.                    if nk>0:
317.                        rs_packet=packet + ''.join('0' for x in range(18-nk))
318.                        p = reedsolo.rs_correct_msg(rs_pac-
        ket, 18, 0, 2, [i for i in range(header,(dlength+18))])[0]
319.                        seq = p[0]
320.                        data = p[1:]
321.                    else:
322.                        seq = ord(packet[0])
323.                        data= packet[1:]
324.
325.                    if seq!=seq_last:
326.                        # Remove zeros padded for last packet
327.                        if 0 < seq <= 7:
328.                            data=data[:-seq]
329.
330.                        if file is None:
331.                            buffer.append(data.decode('utf-8'))
332.                        else:
333.                            open(file, "ab+").write(data)
334.
335.                        seq_last=seq
336.
337.                    # Send ACK
338.                    ser.write(ack_pack)
339.
340.                    if seq <= 7:
341.                        terminator(1) # Respond to communication attempts by sen-
        ding out an ACK
342.                        if file is None:
343.                            return b"".join(buffer)
344.                        else:
345.                            print ("File received successfully")
346.                            logging.debug("In function get_data: File recei-
        ved successfully")
347.                            return 1
348.
349.                except (reedsolo.ReedSolomonError, ZeroDivisionError):
350.                    # Send NAK
351.                    ser.write(nak_pack)
352.
353.                    if nk==18: # Maximum FEC bytes reached
354.                        print("In function get_data: Maximum NAK reached")
355.                        logging.critical("In function get_data: Maximum NAK re-
        ached")
356.                        terminator(0) # Respond to communication attempts by sen-
        ding out a NAK
357.                        return None
358.
359.            # Read header
360.            while True:
361.                header = hread(TOUT_recv)
362.
363.                if header==None:
364.                    print("In function get_data: Connection timed out")
365.                    logging.critical("In function get_data: Connection ti-
        med out")
366.                    return None
367.                elif header=='Error':
368.                    ser.flushInput()
369.                    ser.write(nak_pack)
370.                    continue
371.                elif 6 < header < 9:
372.                    # It is an ACK or a NAK
373.                    print("In function get_data: Expected data, ARQ res-
        ponse found")
```

```
374.                        logging.critical("In function get_data: Expec-
     ted data, ARQ response found")
375.                        return None
376.                else:
377.                    break
378.
379.        # Loop to test the timeout on the send function
380.        def send_loop_tout(packet):
381.            tout_count=0
382.            while True:
383.                # Send packet
384.                ser.write(packet)
385.
386.                # Get the answer from the AUV
387.                while True:
388.                    answer = hread(TOUT_send)
389.                    if answer==None:
390.                        tout_count+=1
391.                        if tout_count > 1:
392.                            updater(0)
393.
394.                        # Test for timeout count
395.                        if tout_count>=2:
396.                            return None
397.                        else:
398.                            break
399.                    else:
400.                        return answer
401.
402.        # Send function:
403.        # stream: data stream to send
404.        # length: length of the object
405.        def send(stream, length):
406.            # Initializing the stream
407.            stream=cStringIO.StringIO(stream)
408.
409.            # Sequence init
410.            seq = 20
411.
412.            # FEC rate calculation and packet distribution from error map
413.            fec_rate=get_fec()
414.            updater(-1, fec_rate)
415.
416.            # Divide the data into packets.
417.            # As the maximum FEC bytes allowed in a packet is 18, the ma-
     ximum fec bytes to add to a 8+1
418.            # message is 18 (corrects 9 errors, fec rate becomes 1).
419.            # With the same reasoning, on a 64+1 packet, the ma-
     ximum code rate is 0.13, and for
420.            # a 236+1 packet it becomes 0.03. However, limitati-
     ons should be done in order to prevent
421.            # errors with the ARQ system, so the code rate should be lo-
     wer than the actual maximum to allow
422.            # NAKs to be used.
423.            data236 = 0.0
424.            data64 = 0.0
425.            data8 = 0.0
426.            if fec_rate < lim236:
427.                data236 = math.floor(length/236)
428.                data64 = math.floor((length%236)/64.0)
429.                data8 = math.ceil(((length%236)%64)/8.0)
430.            elif fec_rate < lim64:
431.                data64 = math.floor(length/64.0)
432.                data8 = math.ceil((length%64)/8.0)
433.            else:
434.                data8 = math.ceil(length/8.0)
```

```python
435.
436.             # Send packets loop
437.             while True:
438.                 # Declare data packet sizes
439.                 if data236 > 0:
440.                     data = stream.read(236)
441.                     data236 -= 1
442.                 elif data64 > 0:
443.                     data = stream.read(64)
444.                     data64 -= 1
445.                 elif data8 > 0:
446.                     data = stream.read(8)
447.                     data8 -= 1
448.
449.                 # Last 8 bytes
450.                 if (data236+data64+data8) == 0:
451.                     # For last packet, sequence number = 8 - len(data)
452.                     seq = 8 - len(data)
453.                     data = data + ((8 - len(data))*'\x00')
454.                 else:
455.                     return 1
456.                 # From this block get: data / len(data)
457.
458.                 # Alternate sequence number if not last packet
459.                 if seq >= 9:
460.                     seq=8
461.                 elif seq == 8:
462.                     seq=9
463.                 # From this block get: seq (1 byte)
464.
465.                 # How many FEC bytes to add to the packet? FEC must be multi-
    ples of 2 for error correction
466.                 fec_bytes = NK(fec_rate, (len(data)+1)) + updater_count
467.
468.                 if fec_bytes > 18:
469.                     fec_bytes = 18
470.                 elif fec_bytes < 2:
471.                     fec_bytes = 2
472.
473.                 #Init vars
474.                 nak_count = 0
475.                 first_packet = []
476.
477.                 #NAK loop
478.                 while True:
479.                     # Send two bytes of FEC for every count of NAK
480.                     if nak_count > 0:
481.                         fec_bytes+=2
482.
483.                     # Detect maximum and minimum of fec bytes
484.                     if fec_bytes == 0:
485.                         fec_bytes = 2
486.                     elif fec_bytes > 18:
487.                         logging.critical('In function send(): FEC bytes re-
    ached maximum')
488.                         print ("In function send(): FEC bytes already maximum")
489.                         return None
490.                     # From this block get: fec_bytes (number of fec_by-
    tes to add)
491.
492.                     # Encode packet with fec_bytes bytes
493.                     # Pre-process Reed-Solomon packet with full 18 bytes
494.                     if nak_count > 0:
495.                         # Send only the two bytes needed (Hybrid ARQ IR)
496.                         fec_packet = first_packet[-(18-fec_bytes+2):-(18-fec_by-
    tes)] if fec_bytes < 18 else first_packet[-2:]
```

```
497.                      else:
498.                          first_packet = reedsolo.rs_en-
     code_msg(chr(seq)+data, 18, 0, 2, gen[18])
499.                      if fec_bytes == 18:
500.                          fec_packet = first_packet
501.                      else:
502.                          fec_packet = first_packet[0:-(18-fec_bytes)]
503.                      # From this block get: packet (final packet con-
     taining seq + msg + cobs + fec
504.
505.                      # Add header according to packet (alternate as the seq num-
     ber for retransmissions)
506.                      packet=reedsolo.rs_encode_msg(chr(len(fec_pac-
     ket)+(nak_count%2)), 2, 0, 2, gen[2])
507.                      packet.extend(fec_packet)
508.
509.                      #sim
510.                      print("Packet to be sent: %s\n" % repr(packet))
511.
512.                      # Sending of packet and timeout checking
513.                      answer = send_loop_tout(packet)
514.                      if answer == None:
515.                          # Header returned None
516.                          logging.critical("In function send(): Max ti-
     meouts while getting response.")
517.                          print("Timeout while sending data")
518.                          return None
519.                      elif answer == 'Error':
520.                          # Header undecodable
521.                          nak_count+=1
522.                          if nak_count > 1:
523.                              updater(0)
524.                          continue
525.
526.                      # Parse and decode answer - if undecodable resend packet
527.                      if answer == ord(ACK):
528.                          if nak_count == 0:
529.                              # Update only if it's within the various pac-
     kets' range
530.                              if (fec_rate < lim236 and data236 > 0) or (lim236 <= f
     ec_rate < lim64 and data64 > 0) or (fec_rate >= lim64):
531.                                  updater(1)
532.                          break
533.                      elif answer == ord(NAK):
534.                          nak_count+=1
535.                          if nak_count == 1:
536.                              # Update only if it's within the various pac-
     kets' range
537.                              if (fec_rate < lim236 and data236 > 0) or (lim236 <= f
     ec_rate < lim64 and data64 > 0) or (fec_rate >= lim64):
538.                                  updater(0)
539.                          else:
540.                              if nak_count == 0:
541.                                  # Update only if it's within the various pac-
     kets' range
542.                                  if (fec_rate < lim236 and data236 > 0) or (lim236 <= f
     ec_rate < lim64 and data64 > 0) or (fec_rate >= lim64):
543.                                      updater(1)
544.
545.                      return answer
546.
547.          print("In function send(): Unexpected error occured")
548.          logging.critical("In function send(): Unexpected error occured")
549.          return None
550.
551.      def location():
```

```
552.              global loc
553.
554.              # Get the current AUV location and store in global vars
555.              location=get_data(send("gl", 2))
556.              if location!=0 and location!=1 and location!=None:
557.                  lst=location.split()
558.              else:
559.                  return 0
560.
561.              loc[0]=int(lst[0])
562.              loc[1]=int(lst[1])
563.
564.              if loc == [None, None]:
565.                  logging.debug("> Location set: Height: N/A Angle: N/A")
566.                  print("Height: N/A Angle: N/A")
567.              else:
568.                  logging.debug("> Location set: Height: %s, An-
       gle: %s" %(loc[0], loc[1]))
569.                  print("Height: %s, Angle: %s" %(loc[0], loc[1]))
570.
571.              return 1
572.
573.         # CMD shell - Gets user input command and parses it infinitely
574.         # Options implemented:
575.         # send [msg] - sends a command message (example of motor control)
576.         # location - request AUV location (example of error map con-
       trol and other uses)
577.         # getfile [file] - request file from AUV (example of longer use of commu-
       nication - can request logs)
578.         class OCP(cmd.Cmd):
579.             intro = 'Welcome to the optical communication proto-
       col shell.   Type help or ? to list commands.\n'
580.             prompt = '> '
581.
582.             def do_send(self, *arg):
583.                 'send [msg]'
584.                 logging.debug('> send %s' % arg)
585.
586.                 if location():
587.                     send("sm " + arg[0], 3+len(arg[0]))
588.                 else:
589.                     print("Failed to retrieve AUV location")
590.                     return
591.
592.             def do_location(self, arg):
593.                 'location'
594.                 logging.debug('> location')
595.                 if location()==0:
596.                     print("Failed to retrieve AUV location")
597.                     return
598.
599.             def do_getfile(self, arg):
600.                 'getfile [file]'
601.                 logging.debug('> getfile %s' % arg)
602.                 if location():
603.                     #sim
604.                     start= time.time()
605.                     get_data(send("fg " + arg, 3+len(arg)), arg)
606.                     print (time.time()-start)
607.                 else:
608.                     print("Failed to retrieve AUV location")
609.                     return
610.
611.             def emptyline(self):
612.                 pass
613.
```

```
614.    def main():
615.        if init()==0:
616.            shutdown()
617.
618.        OCP().cmdloop()
619.
620.    if __name__ == '__main__':
621.        try:
622.            main()
623.        except KeyboardInterrupt:
624.            shutdown()
```

Slave (slave.py) – Changed functions

main():

```python
1.  def main():
2.      if init()==0:
3.          shutdown()
4.
5.      while True:
6.          # Read indefinitely
7.          reader= hread(0)
8.          if reader==0:
9.              ser.flushInput()
10.             continue
11.
12.         # Read data
13.         data=get_data(reader)
14.
15.         if data==None:
16.             continue
17.         else:
18.             data=data.split() # Get command
19.
20.         # Parse data
21.         if data[0]=='gl':
22.             tosend=str(loc[0]) + " " + str(loc[1])
23.             send(tosend, len(tosend))
24.         elif data[0]=='fg':
25.             if data[1]!=None and os.path.isfile(data[1]):
26.                 send(open(data[1], 'rb'), os.stat(data[1]).st_size, type=1)
27.             else:
28.                 continue
29.         elif data[0]=='sm':
30.             # ACK needs to be sent, as there is no DATA to send
31.             ser.write(ack_pack)
32.             terminator(1)
33.             print('Message sent from master: %s' % ' '.join(data[1:]))
34.             logging.debug('Message sent from master: %s' % ' '.join(data[1:]))
35.         else:
36.             terminator(1) # Send ACK
```

get_data():

```python
1.  def get_data(header):
2.      seq_last=int(100) # Random number different from 1
3.      buffer=[]
4.      retr_seq = 0 # Retransmission sequence
5.
6.      while True:
7.          if header=='Error':
8.              ser.flushInput()
9.              ser.write(nak_pack)
10.             header=hread(TOUT_recv)
11.             continue
12.         elif 6 < header < 9:
13.             # It is an ACK or a NAK
14.             print("In function get_data: Expected data, ARQ response found")
15.             logging.critical("In function get_data: Expected data, ARQ res-
    ponse found")
16.             return None
17.         else:
18.             break
19.
```

```
20.     while True:
21.         temp = pread(header)
22.         if temp==None:
23.             print("In function get_data: Expected packet")
24.             logging.critical("In function get_data: Expected packet")
25.             return None
26.
27.         # Check if it's a retransmission
28.         if 2 <= header <= 3:
29.             # It's a retransmission
30.             if retr_seq != header: # Append the FEC symbols only if the se-
    quence is different from the last one
31.                 packet += temp
32.                 header = len(packet)
33.         else:
34.             packet = temp
35.
36.         # Calculate N-K based on packet length
37.         if header >= 237: # 236+1 packet
38.             dlength = 237
39.         elif 65 <= header < 237: # 64+1 packet
40.             dlength = 65
41.         else: # 8+1 packet
42.             dlength = 9
43.
44.         nk = header-dlength
45.
46.         # header = data + current FEC
47.         # dlength = data
48.         # nk = FEC
49.         # dlength + 18 = max packet
50.         try:
51.             # Parse sequence number and data
52.             if nk>0:
53.                 rs_packet=packet + ''.join('0' for x in range(18-nk))
54.                 p = reedsolo.rs_correct_msg(rs_pac-
    ket, 18, 0, 2, [i for i in range(header,(dlength+18))])[0]
55.                 seq = p[0]
56.                 data = p[1:]
57.             else:
58.                 seq = ord(packet[0])
59.                 data= packet[1:]
60.
61.             if seq!=seq_last:
62.                 # Remove zeros padded for last packet
63.                 if 0 < seq <= 7:
64.                     data=data[:-seq]
65.
66.                 buffer.append(data.decode('utf-8'))
67.                 seq_last=seq
68.
69.             if seq <= 7:
70.                 return b"".join(buffer)
71.
72.             # Reset retransmission sequence
73.             retr_seq = 0
74.             # Send ACK
75.             ser.write(ack_pack)
76.
77.         except (reedsolo.ReedSolomonError, ZeroDivisionError):
78.             # Send NAK
79.             ser.write(nak_pack)
80.
81.             if nk==18: # Maximum FEC bytes reached
82.                 print("In function get_data: Maximum NAK reached")
83.                 logging.critical("In function get_data: Maximum NAK reached")
```

```python
84.                terminator(0) # Respond to communication attempts by sen-
    ding out a NAK
85.                return None
86.
87.         # Read header
88.         while True:
89.             header = hread(TOUT_recv)
90.
91.             if header==None:
92.                 print("In function get_data: Connection timed out")
93.                 logging.critical("In function get_data: Connection timed out")
94.                 return None
95.             elif header=='Error':
96.                 ser.flushInput()
97.                 ser.write(nak_pack)
98.                 continue
99.             elif 6 < header < 9:
100.                    # It is an ACK or a NAK
101.                    print("In function get_data: Expected data, ARQ res-
    ponse found")
102.                    logging.critical("In function get_data: Expec-
    ted data, ARQ response found")
103.                    return None
104.                else:
105.                    break
```

send():

```python
1.  def send(stream, length, type=0):
2.      # Initializing the stream
3.      if not type:
4.          stream=cStringIO.StringIO(stream)
5.
6.      # Sequence init
7.      seq = 20
8.
9.      # FEC rate calculation and packet distribution from error map
10.     fec_rate=get_fec()
11.     updater(-1, fec_rate)
12.
13.     # Divide the data into packets.
14.     # As the maximum FEC bytes allowed in a packet is 18, the maximum fec by-
    tes to add to a 8+1
15.     # message is 18 (corrects 9 errors, fec rate becomes 1).
16.     # With the same reasoning, on a 64+1 packet, the ma-
    ximum code rate is 0.13, and for
17.     # a 236+1 packet it becomes 0.03. However, limitations should be done in or-
    der to prevent
18.     # errors with the ARQ system, so the code rate should be lower than the ac-
    tual maximum to allow
19.     # NAKs to be used.
20.     data236 = 0.0
21.     data64 = 0.0
22.     data8 = 0.0
23.     if fec_rate < lim236:
24.         data236 = math.floor(length/236)
25.         data64 = math.floor((length%236)/64.0)
26.         data8 = math.ceil(((length%236)%64)/8.0)
27.     elif fec_rate < lim64:
28.         data64 = math.floor(length/64.0)
29.         data8 = math.ceil((length%64)/8.0)
30.     else:
31.         data8 = math.ceil(length/8.0)
32.
33.     # Send packets loop
```

```python
34.     while True:
35.         # Declare data packet sizes
36.         if data236 > 0:
37.             data = stream.read(236)
38.             data236 -= 1
39.         elif data64 > 0:
40.             data = stream.read(64)
41.             data64 -= 1
42.         elif data8 > 0:
43.             data = stream.read(8)
44.             data8 -= 1
45.
46.             # Last 8 bytes
47.             if (data236+data64+data8) == 0:
48.                 # For last packet, sequence number = 8 - len(data)
49.                 seq = 8 - len(data)
50.                 data = data + ((8 - len(data))*'\x00')
51.         else:
52.             return 1
53.         # From this block get: data / len(data)
54.
55.         # Alternate sequence number if not last packet
56.         if seq >= 9:
57.             seq=8
58.         elif seq == 8:
59.             seq=9
60.         # From this block get: seq (1 byte)
61.
62.         # How many FEC bytes to add to the packet? FEC must be multi-
    ples of 2 for error correction
63.         fec_bytes = NK(fec_rate, (len(data)+1)) + updater_count
64.
65.         if fec_bytes > 18:
66.             fec_bytes = 18
67.         elif fec_bytes < 2:
68.             fec_bytes = 2
69.
70.         #Init vars
71.         nak_count = 0
72.         first_packet = []
73.
74.         #NAK loop
75.         while True:
76.             # Send two bytes of FEC for every count of NAK
77.             if nak_count > 0:
78.                 fec_bytes+=2
79.
80.             # Detect maximum and minimum of fec bytes
81.             if fec_bytes == 0:
82.                 fec_bytes = 2
83.             elif fec_bytes > 18:
84.                 logging.critical('In function send(): FEC bytes reached ma-
    ximum')
85.                 print ("In function send(): FEC bytes already maximum")
86.                 return None
87.             # From this block get: fec_bytes (number of fec_bytes to add)
88.
89.             # Encode packet with fec_bytes bytes
90.             # Pre-process Reed-Solomon packet with full 18 bytes
91.             if nak_count > 0:
92.                 # Send only the two bytes needed (Hybrid ARQ IR)
93.                 fec_packet = first_packet[-(18-fec_bytes+2):-(18-fec_by-
    tes)] if fec_bytes < 18 else first_packet[-2:]
94.             else:
95.                 first_packet = reedsolo.rs_en-
    code_msg(chr(seq)+data, 18, 0, 2, gen[18])
```

```
96.                 if fec_bytes == 18:
97.                     fec_packet = first_packet
98.                 else:
99.                     fec_packet = first_packet[0:-(18-fec_bytes)]
100.                 # From this block get: packet (final packet con-
     taining seq + msg + cobs + fec
101.
102.                 # Add header according to packet (alternate as the seq num-
     ber for retransmissions)
103.                 packet=reedsolo.rs_encode_msg(chr(len(fec_pac-
     ket)+(nak_count%2)), 2, 0, 2, gen[2])
104.                 packet.extend(fec_packet)
105.                 # Sending of packet and timeout checking
106.                 answer = send_loop_tout(packet)
107.                 if answer == None:
108.                     # Header returned None
109.                     logging.critical("In function send(): Max ti-
     meouts while getting response.")
110.                     print("Timeout while sending data")
111.                     return None
112.                 elif answer == 'Error':
113.                     # Header undecodable
114.                     nak_count+=1
115.                     if nak_count > 1:
116.                         updater(0)
117.                     continue
118.
119.                 # Parse and decode answer - if undecodable resend packet
120.                 if answer == ord(ACK):
121.                     if nak_count == 0:
122.                         # Update only if it's within the various pac-
     kets' range
123.                         if (fec_rate < lim236 and data236 > 0) or (lim236 <= f
     ec_rate < lim64 and data64 > 0) or (fec_rate >= lim64):
124.                             updater(1)
125.                     break
126.                 elif answer == ord(NAK):
127.                     nak_count+=1
128.                     if nak_count == 1:
129.                         # Update only if it's within the various pac-
     kets' range
130.                         if (fec_rate < lim236 and data236 > 0) or (lim236 <= f
     ec_rate < lim64 and data64 > 0) or (fec_rate >= lim64):
131.                             updater(0)
```

Reed-Solomon simulations (sim_rs.py)

```python
1.  import reedsolo
2.  import random
3.  import matplotlib
4.  matplotlib.use("Pdf")
5.  import matplotlib.pyplot as plt
6.
7.  # Factorial of a number
8.  def factorial(n):return reduce(lambda x,y:x*y,[1]+range(1,n+1))
9.
10. # Calculation of packet decoding probability
11. def PDP(p, N, nk):
12.     min = (nk/2)+1
13.     pdp=0
14.     for k in range(min, N+1):
15.         # Calculate PDP with the formula from literature
16.         pdp += (float(factorial(N))/(factorial(k)*factorial(N-k)))*(p**k)*((1-
    p)**(N-k))
17.
18.     return 1-pdp
19.
20. # Init
21. reedsolo.init_tables(0x11d)
22.
23. # Probability of errors
24. #pe = [0.1, 0.25, 0.5, 0.75, 1]
25. pe = [0.001, 0.01, 0.03, 0.05, 0.08, 0.1, 0.15, 0.25, 0.40, 0.5]
26. petrace=pe+[1]
27. # Number of packets to test for each probability
28. n= 10000
29.
30. #nkarr= [2]
31. nkarr= [2,8,18]
32.
33. for nfig in range(len(nkarr)):
34.     # Packet to be tested
35.     nk=nkarr[nfig]
36.     packet=reedsolo.rs_encode_msg('123456789', nk)
37.     #packet=reedsolo.rs_encode_msg('1', nk)
38.     l=len(packet)
39.
40.     dec=[0] * (len(pe)+1)
41.     ndec=[0] * (len(pe)+1)
42.     idec=[0] * (len(pe)+1)
43.
44.     # Simulation loop
45.     for i, inum in enumerate(pe):
46.         print('Progress: %d/%d' % (i+1, len(pe)))
47.         for j in range(n):
48.             en = packet[:]
49.             # Channel simulation
50.             for x,y in enumerate(en):
51.                 # Add errors based on probability to be tested
52.                 if random.random() < pe[i]:
53.                     # Avoid writing a symbol that is the same as the original
54.                     if en[x]!=1:
55.                         en[x]=1
56.                     else:
57.                         en[x]=0
58.
59.             if en[:-nk] == packet[:-nk]:
60.                 ndec[i]+=1
61.
62.             # Test if decodable
```

```python
63.            try:
64.                de = reedsolo.rs_correct_msg(en, nk)[0]
65.                # Correct decoding?
66.                if de == packet[:-nk]:
67.                    dec[i]+=1
68.                else:
69.                    idec[i]+=1
70.            except reedsolo.ReedSolomonError:
71.                continue
72.
73.      # Get theoretical values
74.      # Probabilities of decoding
75.      pt = [PDP(m, l, nk) for y,m in enumerate(pe)]
76.      # Multiply by packets
77.      dect= [a*n for y,a in enumerate(pt)] + [0]
78.
79.      # ndec - Uncoded number
80.      # dec - Decoded number
81.      # dect - Theoretical value
82.      # idec - Incorrect decoding
83.      print dec
84.
85.      # Plotting 1
86.      print ('Plotting...')
87.      fig, ax = plt.subplots()
88.      ax.plot(petrace, [(ndec[plt1]/float(n))*100 for plt1 in range(len(petrace))],
    'k--', label="Uncoded")
89.      ax.plot(petrace, [(dect[plt2]/float(n))*100 for plt2 in range(len(petrace))],
    'k:', label="Theory")
90.      ax.plot(petrace, [(dec[plt3]/float(n))*100 for plt3 in range(len(petrace))],
    'kx', label="Experimental")
91.      ax.legend(loc=0)
92.      plt.axis([petrace[0],petrace[-1] , 0, 100])
93.      plt.xscale('log')
94.      plt.yscale('linear')
95.      plt.xlabel('P(symbol error)')
96.      plt.ylabel('Decoded packets (%)')
97.      plt.grid(True)
98.      print ('Saving...')
99.      plt.savefig(('sim-rs-%d.jpg' % nfig), bbox_inches='tight')
100.           #plt.savefig(('sim-rs-header-arq-
    %d.jpg' % nfig), bbox_inches='tight')
101.
102.           # Plotting 1
103.           print ('Plotting...')
104.           fig, ax = plt.subplots()
105.           ax.plot(petrace, [(idec[plt4]/float(n))*100 for plt4 in range(len(petr
    ace))], 'k-', label="Inc decodings")
106.           plt.axis([petrace[0], petrace[-1], 0, 20])
107.           plt.xscale('log')
108.           plt.yscale('linear')
109.           plt.xlabel('P(symbol error)')
110.           plt.ylabel('Undetected errors (%)')
111.           plt.grid(True)
112.           print ('Saving...')
113.           plt.savefig(('sim-rs-%d-inc.jpg' % nfig), bbox_inches='tight')
114.           #plt.savefig(('sim-rs-%d-header-arq-
    inc.jpg' % nfig), bbox_inches='tight')
```

References

1. Pan-Mook Lee, Bong-Hwan Jeon, and Sea-Moon Kim, "Visual servoing for underwater docking of an autonomous underwater vehicle with one camera," in *Proc. MTS/IEEE OCEANS*, San Diego, 2003, pp. 2195-2200.

2. L. Paull, S. Saeedi, M. Seto, and H. Li, "AUV Navigation and Localization: A Review," *IEEE Journal of Oceanic Engineering*, vol. 39, no. 1, pp. 131-149, Dec. 2013.

3. N. A. Cruz, A. C. Matos, R. M. Almeida, and B. M. Ferreira, "A lightweight docking station for a hovering AUV," *2017 IEEE Underwater Technology (UT)*, Busan, pp. 1-7, Apr. 2017.

4. G. Wang, J. Han, X. Wang, and D. Jing, "Improvement on vision guidance in AUV docking," in *Conf. OCEANS 2016 - Shanghai*, Shanghai, 2016, pp. 1-6.

5. S. Martin et al., "Characterizing the critical parameters for docking unmanned underwater vehicles," in *Proc. OCEANS 2016 MTS/IEEE Monterey*, Monterey, 2016, pp. 1-7.

6. H. Kaushal, and G. Kaddoum, "Underwater Optical Wireless Communication," *IEEE Access*, vol. 4, pp. 1518-1547, Apr. 2016.

7. A. A. Khan, M. H. Rehmani, and M. Reisslein, "Cognitive Radio for Smart Grids: Survey of Architectures, Spectrum Sensing Mechanisms, and Networking Protocols," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 860-898, Sep. 2015.

8. T. Zhang, D. Li, and C. Yang, "Study on impact process of AUV underwater docking with a cone-shaped dock," *Ocean Engineering*, vol. 130, pp. 176–187, Jan. 2017.

9. L. Wu, Y. Li, S. Su, P. Yan, and Y. Qin, "Hydrodynamic analysis of AUV underwater docking with a cone-shaped dock under ocean currents," *Ocean Engineering*, vol. 85, pp. 110–126, Jul. 2014.

11. B. M. C. Silva, "Underwater optical communication: an approach based on LED," 2015, Dissertation.

12. Z. Yue and T. Wang, "Navigation and positioning system design of an AUV underwater docking," in *Conf. 2016 IEEE/OES China Ocean Acoustics (COA)*, Harbin, 2016, pp. 1-6.

13. N. Farr, A. Bowen, J. Ware, C. Pontbriand, and M. Tivey, "An integrated, underwater optical /acoustic communications system," in *Conf. OCEANS 2010 IEEE - Sydney*, Sydney, 2010, pp. 1-6.

14. Y. Sato, T. Maki, K. Masuda, T. Matsuda, and T. Sakamaki, "Autonomous docking of hovering type AUV to seafloor charging station based on acoustic and visual sensing," in *Conf. 2017 IEEE Underwater Technology (UT)*, Busan, 2017, pp. 1-6.

15. Unmounted LEDs [Internet]. [accessed 2017 Oct 10]. Available from: https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=2814

16. Si Avalanche Photodetectors [Internet]. [accessed 2017 Oct 10]. Available from: https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=6686

17. Unmounted Photodiodes [Internet]. [accessed 2017 Oct 10]. Available from: https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=285

18. P. Góis et al., "Development and validation of blue ray, an optical modem for the MEDUSA class AUVs," in *Conf. 2016 IEEE Third Underwater Communications and Networking Conference (UComms)*, Lerici, 2016, pp. 1-5.

19. D. Wen, W. Cai, and Y. Pan, "Design of underwater optical communication system," in *Conf. OCEANS 2016 - Shanghai*, Shanghai, 2016, pp. 1-4.

20. S. Hu, L. Mi, T. Zhou, and W. Chen, "Viterbi equalization for long-distance, high-speed underwater laser communication," *Optical Engineering*, vol. 56, no. 7, p. 076101, Jun. 2017.

21. Water absorption spectrum [Internet]. [accessed 2018 Jan 11]. Available from: http://www1.lsbu.ac.uk/water/water_vibrational_spectrum.html#d

22. Shu Lin, D. J. Costello, and M. J. Miller, "Automatic-repeat-request error-control schemes," *IEEE Communications Magazine*, vol. 22, no. 12, pp. 5-17, Dec. 1984.

23. M. Tomlinson, C. J. Tjhai, M. A. Ambroze, M. Ahmed, and M. Jibril, "Error-Correction Coding and Decoding," *Signals and Communication Technology*, 2017.

24. B. Tahir, S. Schwarz, and M. Rupp, "BER comparison between Convolutional, Turbo, LDPC, and Polar codes," in *Conf. 2017 24th International Conference on Telecommunications (ICT)*, Limassol, 2017, pp. 1-7.

25. S. Kumar, and R. Gupta, "Performance comparison of different forward error correction coding techniques for wireless communication systems," *International Journal of Computer Science and Technology (IJCST)*, vol. 2, no. 3, pp. 553-557, 2011.

26. L. Lopacinski, J. Nolte, S. Buechner, M. Brzozowski, and R. Kraemer, "Improved turbo product coding dedicated for 100 Gbps wireless terahertz communication," in *Conf. 2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, Valencia, 2016, pp. 1-6.

27. I. S. Reed, and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

28. W. C. Cox, J. A. Simpson, C. P. Domizioli, J. F. Muth, and B. L. Hughes, "An underwater optical communication system implementing Reed-Solomon channel coding," in *Conf. OCEANS 2008, Quebec City*, Quebec City, 2008, pp. 1-6.

29. J. A. Simpson, W. C. Cox, J. R. Krier, B. Cochenour, B. L. Hughes, and J. F. Muth, "5 Mbps optical wireless communication with error correction coding for underwater sensor nodes," in *Conf. OCEANS 2010 MTS/IEEE SEATTLE*, Seattle, 2010, pp. 1-4.

30. P. Shrivastava, and U. P. Singh, "Error detection and correction using Reed Solomon Codes," *International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE)*, vol. 3, no. 8, pp. 965-969, Aug. 2013.

31. S. Choi, and K. G. Shin, "A class of adaptive hybrid ARQ schemes for wireless links," *IEEE Transactions on Vehicular Technology*, vol. 50, no. 3, pp. 777-790, May 2001.

32. R. Diamant, and L. Lampe, "Adaptive Error-Correction Coding Scheme for Underwater Acoustic Communication Networks," *IEEE Journal of Oceanic Engineering*, vol. 40, no. 1, pp. 104-114, Jan. 2014.

33. L. Ma, and Y. Wei, "An Incremental Redundancy HARQ Scheme for Polar Code," *arXiv preprint*, arXiv:1708.09679, 2017.

34. H. Schumny, "ECMA 1983/84," *Computers and Standards*, vol. 3, no. 3-4, pp. 199-206, 1984.

35. W. Simpson, "RFC 1662: PPP in HDLC-like Framing," 1994.

36. S. Cheshire, and M. Baker, "Consistent overhead byte stuffing," *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 159-172, Apr. 1999.

37. H. Nasir et al., "CoDBR: Cooperative Depth Based Routing for Underwater Wireless Sensor Networks," in *Conf. 2014 Ninth International Conference on Broadband and Wireless Computing, Communication and Applications*, Guangdong, 2014, pp. 52-57.

38. Raspberry Pi 2, Model B [Internet]. [accessed 2017 Oct 10]. Available from: https://cdn-shop.adafruit.com/pdfs/raspberrypi2modelb.pdf

39. T. Filiba, "reedsolomon: A pure-python Reed Solomon encoder/decoder," *GitHub*, GitHub Repository, 2017. Available from: https://github.com/tomerfiliba/reedsolomon.

40. C. Liechti, "pyserial: Python serial port access library," *GitHub*, GitHub Repository, 2017. Available from: https://github.com/pyserial/pyserial.

41. J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90-95, May 2007.

42. F. Schill, U. R. Zimmer, and J. Trumpf, "Visible spectrum optical communication and distance sensing for underwater applications," *Proc. ACRA*, 2004, pp. 1-8.

43. B. Ferreira, A. Matos, and N. Cruz, "Single beacon navigation: Localization and control of the MARES AUV," in *Conf. OCEANS 2010 MTS/IEEE SEATTLE*, Seattle, 2010, pp. 1-9.

44. C. Grinstead, and J. Snell, "Introduction to probability," *American Mathematical Soc.*, pp. 212-218, 2012.

88 References

Annex - Licenses

Water absorption spectrum

Figure 2.1

As stated on the web page:

```
This work is licensed under a Creative Commons Attribution
-Noncommercial-No Derivative Works 2.0 UK: England & Wales License
```

Reed-Solomon python library

As stated in the LICENSE file of the repository:

```
Released to the public domain. Original implementation can be found at
http://en.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders
```

PySerial

As stated in PySerial's official website (https://pythonhosted.org/pyserial/appendix.html#license):