Applicability of the Julia Programming Language to Forward Error-Correction Coding in Digital Communications Systems

A project
presented to
Department of Computer and Information Sciences
State University of New York Polytechnic Institute
At Utica/Rome
Utica, New York

In partial fulfillment
Of the requirements of the
Master of Science Degree

by
Ryan Quinn
May 2018

Declaration

I declare that this project is my own work and has not been submitted in any form for another degree of diploma at any university of other institute of tertiary education. Information derived from published and unpublished work of others has been acknowledged in the text and a list of references given.

_____

Ryan Quinn

SUNY POLYTECHNIC INSTITUTE

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES

Approved and recommended for acceptance
as a project in partial fulfillment of the requirements
for the degree of Master of Science in
computer and information science

_____

DATE

_____

Dr. Bruno R. Andriamanalimanana
Advisor

_____

Dr. Saumendra Sengupta

_____

Dr. Scott Spetka

1 ABSTRACT

Traditionally SDR has been implemented in C and C++ for execution speed and processor efficiency. Interpreted and high-level languages were considered too slow to handle the challenges of digital signal processing (DSP). The Julia programming language is a new language developed for scientific and mathematical purposes that is supposed to write like Python or MATLAB and execute like C or FORTRAN.

Given the touted strengths of the Julia language, it bore investigating as to whether it was suitable for DSP. This project specifically addresses the applicability of Julia to forward error correction (FEC), a highly mathematical topic to which Julia should be well suited.

It has been found that Julia offers many advantages to faithful implementations of FEC specifications over C/C++, but the optimizations necessary to use FEC in real systems are likely to blunt this advantage during normal use. The Julia implementations generally effected a 33% or higher reduction in source lines of code (SLOC) required to implement. Julia implementations of FEC algorithms were generally not more than 1/3 the speed of mature C/C++ implementations.

While Julia has the potential to achieve the required performance for FEC, the optimizations required to do so will generally obscure the closeness of the implementation and specification. At the current time it seems unlikely that Julia will pose a serious challenge to the dominance of C/C++ in the field of DSP.

2 TABLE OF CONTENTS

3 LIST OF FIGURES

4 LIST OF TABLES

5 LIST OF ABBREVIATIONS

- AM: Amplitude Modulation
- DSP: Digital Signal Processing
- FEC: Forward Error Correction
- FM: Frequency Modulation
- GC: Garbage Collector
- GFP: Generic Framing Procedure
- IDE: Integrated Development Environment
- ITU: International Telecommunications Union
- JNI: Java-Native Interface
- LTS: Long-Term Support
- LUT: Look-up Table
- MDS: Maximum-Distance Seperable (code)
- REPL: Read-Evaluate-Print-Loop
- RS: Reed-Solomon
- SBC: Single-Board Computer
- SDR: Software-Defined Radio
- SLOC: Source Lines of Code
- SSD: Solid-State Drive
- SWIG: Simple Wrapper and Interface Generator

6 BACKGROUND

6.1 OVERVIEW

Communications systems are one of the linchpins of the "Information Age." In a world defined by the rapid and easy transfer and sharing of information over arbitrary distances, communications systems are the backbone. With increased demand for information, there have been unprecedented levels of work on improving the capabilities of communications systems: in 1998, Jakob Nielsen of the Nielsen Norman Group posited what later became known as "Nielsen's Law," a bandwidth-focused corollary of Moore's Law, which predicted that bandwidth would double every 21 months [1]. As of 2016, his law still holds (the most recent data available). With the proliferation of wireless devices in recent years, communications systems are being asked to do more and more with fewer and fewer resources in increasingly contested environments.

Communications systems have always been unreliable, with extensive research into achieving higher data rates and preventing errors dating back to the first half of the 20th century [2] [3] [4]. Unreliability is simply a consequence of trying to send an ordered pattern ("information") through a chaotic world ("entropy"). Early work focused primarily on telegraphy and telephony, which at the time both traveled through a tightly-controlled and solid medium. Introducing wireless communications into the mix adds an additional level of complexity in that one will never know the conditions of the communications "channel" *a priori*. Wireless communication introduces Doppler fading, Rayleigh fading, multipath, Fresnel effects, and channel coherence time concerns, among others. While channel models exist to attempt to estimate the conditions of wireless communications [5], at the end of the day uncertainty is nearly a fact of life for wireless communications.

Because of the presence of uncertainty in communications systems, digital communications have developed countermeasures against errors. These countermeasures, generally referred to as "forward error-correcting codes," or "FEC," are mathematical means of adding some pattern of known digital

information to a message at the transmitter to allow extraction of the original message at the receiver even in the presence of errors.

As an example, consider a wireless text-messaging application. Even when constrained to a length of 180 characters, the number of possible messages that may be transmitted is inconceivably large. Without some means of error correction, the receiver can have no certainty as to whether the message they received matches the transmitted message. For example, the following messages differ by only a single bit in the underlying binary representation:

"There is a 10% chance of rain today."

"There is a 90% chance of rain today."

Despite the similarity of the underlying *data*, the small difference drastically changes the *information* of the message: the recipient is likely to plan significantly differently depending on which message is received. It is important to note that while data and information are often used interchangeably, they have different technical definitions (see **Section 6.2**, below) and engineers tend to use a different concept of information than information scientists [6].

FEC seeks to correct this shortcoming by adding some known pattern to the message at the transmitter to assist in its decoding at the receiver [7]. The more of a known pattern that is added, the more confidence the receiver will have in decoding the message, at the cost of less message (less new information) being transmitted in the same period [6]. Most FEC is a compromise between these two competing demands (integrity and transfer rate).

Early error-correction codes predate their use in digital communications systems. Nyquist makes note of their use in radio telegraphy [3], while Hamming initially proposed his error correction codes as a means of aiding computer programming [7]. Primitive codes used majority-logic voting such as "two-out-of-three" or "three-out-of-five" [8]. These codes could correct one error out of every three or two errors out of every five bits, respectively. Today, they would be said to have a "Hamming

distance" of three and five, respectively (Hamming distance is described in detail later in this document). While effective at correcting errors, these codes have high message overhead, able to transmit data at only ⅓ or ⅕ the maximum possible rate. The first digital system to use a "two-out-of-three" error correction code was the SAPO, the first fault-tolerant computer [9].

Another early form or error detection, the parity bit, was able to detect, but not correct, one bit-error per block of data. Parity bits were used by the Bell Model V computer to detect errors in the punch cards used for programming. If an error was detected, the machine would either alert the user or fail silently, a situation that proved unacceptable to one of the engineers at Bell Laboratories at the time, Richard W. Hamming [10]. Hamming developed a generalized nomenclature for error-correction codes based on patterns between them, eventually leading to the development of what are now known as "Hamming codes" based on linear algebra principles [7].

Error correction coding was first applied to communications systems by Claude Shannon, a colleague of Hamming at Bell Laboratories [11]. The mathematics put forward by Hamming and Shannon applied only to messages of a fixed length but were generalized for messages and codes of arbitrary size within a year by Marcel Golay of the Signal Corps Engineering Laboratory [12]. Since then, thanks to the proliferation of data and the need to protect it over ever more demanding channels (especially wireless and space applications), more powerful (and correspondingly more complex) FEC has been developed over the years to compensate.

Recent years have seen a veritable explosion in the number of wireless devices in use and the amount of data moved between them. IT giant Cisco noted in a 2017 report that mobile data traffic was up 18-fold over the previous five years [13], while CTIA places the number at a 35-fold increase between 2010 and 2016 [14]. Mobile data traffic grew 63% in 2016, with half a billion new devices connected [13]. In the United States, there are now more mobile devices than people [14].

Competition and commoditization pressures have forced rapid advances in wireless communications. While wireless communications hardware has improved greatly over the last two decades, as evidenced by the rapidly increasing throughput requirements of IEEE 802.11 ("Wi-Fi") standards [15] [16], hardware will always have a slower adoption and upgrade cycle than software due to its physical nature. The "Principle of Equivalence of Hardware and Software" [17] states that anything that can be done in hardware can be done in software, and vice versa. Null and Lobur note, however, that hardware implementations are almost always faster. For systems with real-time constraints, which include most communications systems, speed is always a factor. For most of the history of communications systems, even the relatively short history of wireless communications systems, the relatively slow speed of software made it an impractical solution for communications systems.

Of course, in the case of general-purpose computers, fast hardware generally translates to fast software. And as noted by David House in a quote generally attributed to Gordon Moore and popularly known as "Moore's Law," [18] [19] computer speeds have doubled every ~18 months for the last 40 years. Recently many prominent computer scientists have been publicly bearish on the future of Moore's Law, notably Herb Sutter [20] [21] and Moore himself [22].

Regardless of what the future may or may not bring for computer hardware, four decades of exponential growth is not insignificant. In 2011, venture capitalist Marc Andreessen famously declared in the pages of the Wall Street Journal that "software [was] eating the world [23]." Around the same time, the first viable and public implementations of wireless communications standards in software began to emerge from the academic community [24] [25].

Today, software can fill many roles in communications systems traditionally filled by hardware. When used with wireless communications systems, this is referred to as software-defined radio (SDR). Many SDR systems have historically been written in C/C++ for performance purposes, including the

backend of the popular SDR library gnuradio and many of the components of the U.S. military Joint Tactical Radio System (JTRS) [26]. Despite their widespread popularity, C/C++ have never been accused of being easy to write or maintain: in *Code Complete*, author Steve McConnell of Construx Software jokes that "You save time when you don't have to have an awards ceremony every time a C statement does what it's supposed to do." [27] Similar contentions have been made by other prominent voices in the field [28]. At the other end of the spectrum are the so-called "fourth-generation languages," which are generally said to be fast to write and slow to execute. Examples include Python (the frontend of gnuradio) and Tcl [28].

Though they speed up implementation, high-level languages may not provide the performance necessary for real-time systems. In mature SDR platforms, such as gnuradio, an effort was made to expose the "best of both worlds" by allowing low-level DSP code written in C/C++ or FORTRAN to be "wrapped" in Python. One of the most prominent tools for this task is the Simple Wrapper and Interface Generator (SWIG) [29]. In more recent years a new paradigm emerged in the numerical computing and mathematically-inclined segments of software development: was there a reason that code couldn't have both fast execution speeds and be easy to write?

Enter the Julia programming language [30]. Julia purports to be a language that writes like a 4th-generation language and executes like a 3rd-generation language. In terms of SDR, it should fit the bill perfectly: real-time systems and rapid development environments both have time constraints and require speed. Perhaps Julia is the correct tool for the job.

Given the breadth of the topic of "software-defined radio," the remainder of this report will focus on one specific aspect of communications systems: forward error-correction coding (FEC). FEC is still an area of active research and development with applicability to every modern form of communication. Given its affinity for numerical computing, the Julia programming language should prove a valuable addition to the implementation of FEC in software.

6.2 INFORMATION: A DEFINITION

Since FEC is in the business of preserving information, it is necessary to first define what information is and is not. Information is not data [31] (this point is often belabored by information scientists), but it can be created from data. A precise definition of information is the subject of much debate [6].

Perhaps the easiest definition of information to understand is the Bateson definition [32]: "information is any difference that makes a difference in a conscious human mind." That definition is perhaps more pithy than precise, however, and the Parker definition may be more appropriate in an engineering context [33]: "information is the pattern of organization of matter and energy." This is the definition commonly accepted by information scientists, though a comprehensive discussion of the topic is available in [34].

In his landmark 1948 paper on the topic [11], Claude Shannon posited the following theory of information: information was data transferred between two parties, and this transfer may fail. The data must be a set of symbols selected from some finite universe of symbols agreed upon by the sender and receiver. The more *a priori* information the receiver has about the message, the easier it will be to receive, and the less effect entropy will have on the message.

The Shannon Model of information was designed only to apply to communications systems, though it has been applied (and misapplied) elsewhere for nearly as long as it has existed [34]. One of the key confusions of the Shannon Model is this: Shannon posited that more uncertainty in a message means more information may transferred [11]. In his view, the easiest messages to receive are highly structured messages well-known to the receiver. These messages are easily decoded, but to use the Bateson definition of information, they are less likely to make a difference to the human mind at the receiver, as they are already familiar. In contrast, it may be difficult for the receiver to decode a brand-

new message that contains only never-before-seen information, though all the information (correct or not) would “make a difference” to the receiver.

In the realm of communication systems, the Shannon model is a key tool in understanding the tradeoffs of FEC (see **Section 6.5**, below). Outside the realm of communications systems, however, the Shannon Model falters. For instance, Miller [35] notes that according to the Shannon Model, the sentence “Rex is a mammal” contains more information than “Rex is a dog.” Intuitively this is nonsensical: “dog” is a strict subset of “mammal” and thus anything that is true of mammals is true of dogs, *plus an additional well-defined set of traits.*

6.3 COMMUNICATIONS SYSTEMS

A communications system is a system for disseminating information [36]. Prototypical examples include the radio, telephone, and Internet, but newspapers, billboards, and books are also communications systems. At first blush the latter two examples may appear better classified as storage systems instead of communications systems, but the difference between the two is entirely semantic. Indeed, it has been noted [37] that storage systems are akin to “communication through time” and thus suffer from many of the same challenges as communications systems. By the same token, many techniques applied to communications systems are also relevant to storage systems, including compression coding and error correction coding.

The most iconic description of a communications system is attributed to Claude Shannon [11] and reprinted below:

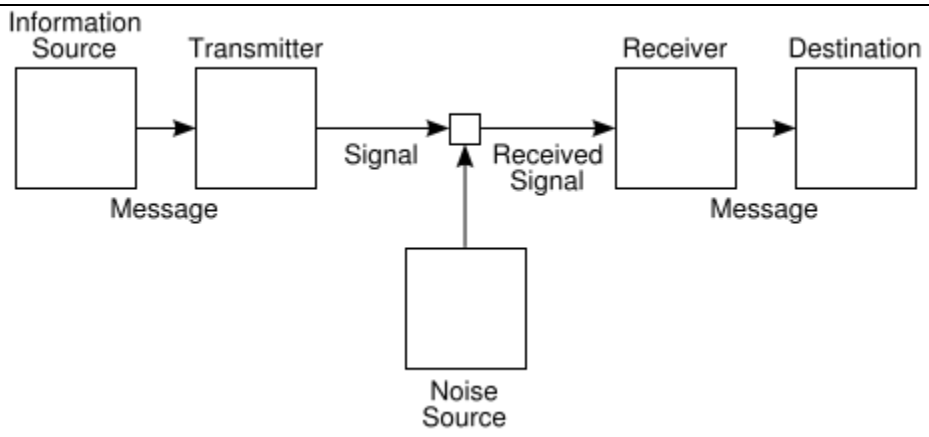

Figure 1: The Shannon Model of a communications system. *Image is in the public domain, original source* [11].

In the Shannon Model, information is moved from a sender to a receiver through some noisy "channel," where the channel may be a telegraph wire, the air, or in the case of a storage system, time. The nature of the channel is immaterial to the modeling of the system. "Noise" here varies by context: for wireless communications it is generally a combination of Gaussian noise, interference, Doppler effects, Raleigh fading, and ground effects; but for a book it might simply be the ravages of everyday use. FEC is intended to reduce or control the effects of the channel on the original information.

6.4 Basic Principles of Digital Communication

Digital communication is a topic far too broad to be fully addressed in a report of this size. For the sakes of both completeness and perspective, however, a brief overview is provided below.

6.4.1 Overview

Communications systems have existed for as long as there has been a means of representing information in a physical form. In the most general sense, sending a letter via a courier or through the mail is a "communications system:" information is disseminated through a channel (the USPS) by a transmitter (the writer) and acquired by the receiver (the reader). Over time, communications systems became more complex, eventually encompassing radio, telegraphy, and telephony. Many of these

communications advancements rank among the most important of the 20th century, according to the United States Academy of Engineers [38].

All communications systems mentioned above have one common trait, however. They are all means of transmitting analog information through an analog channel by analog means. For example, frequency-modulated (FM) radio uses a sensor that translates the pitch and amplitude of the human voice into an electrical signal. This signal it sent (almost) directly to an antenna for transmission [39]. On the receiver, the antenna has an impulse response to incoming signals that varies by frequency. When the FM radio waves arrive, their modulation interacts with the varying antenna response to produce an electric current that feeds (almost) directly into the speakers of the receiver, which can convert it back to sound. There is no need to correlate or align or clock the signal. The only two states for the receiver are "no signal" (white noise) and "signal exists" (and thus sound).

Digital communications are distinctly different animals, with different benefits and drawbacks compared to analog signals. Digital data is discrete; it can only have one of two states (one or zero), and a combination of these states allows it to approximate the analog world [40]. Analog data, by contrast, can have any number of states. It can never be fully represented in the digital world. Devices such as digital-to-analog and analog-to-digital converters (DAC and ADC, respectively) exist to help bridge the gap between the two worlds, but the gap will always exist. So, the question remains: how is digital information represented in the analog world?

The most iconic representation of a digital communications system is the Sklar model [41] and is shown below in **Figure 2**. "Source encoding," sometimes referred to as framing, is the link between higher-level protocols and the communications system. In the OSI model, the communications system is a Layer 1 (PHY) protocol, so it needs some interface to the link layer above it. Encryption is not strictly necessary and is beyond the scope of this paper. "Channel encoding" is another term for FEC, which is discussed at length in the next section. Multiplexing allows the system to combine multiple higher-level

channels and is not always necessary. Modulation serves as the link between the digital communications system and the analog world and is discussed in brief below. Frequency-spread and multiple access are specialized techniques beyond the scope of this document.

Note that at a high level, the transmission and reception processes are symmetrical.

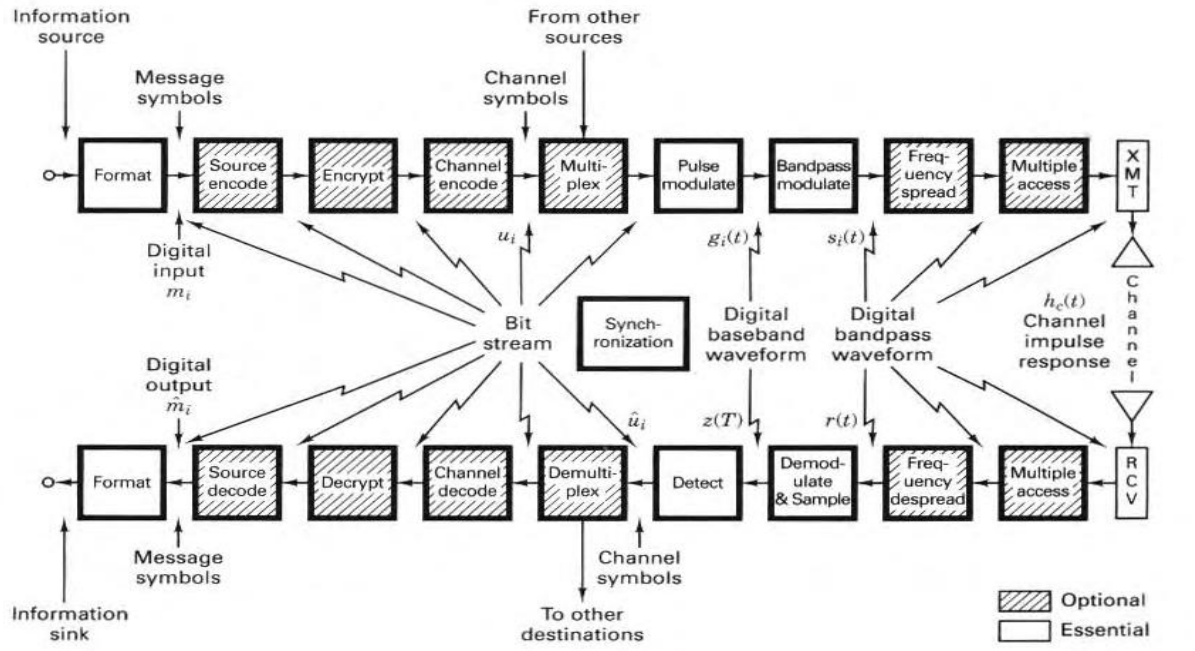

Figure 2: The Sklar diagram, a high-level overview of digital communication systems [41].

6.4.2 Framing/Source Encoding

Digital communications systems are Level-1 protocols. As such, they need some way to map between higher-layer protocols and their own internal representation of the data. This usually involves wrapping the data frame in a header and some means of error detection. The header contains some manner of synchronization code to ensure that the data are bit-aligned, the necessary protocol information, and the payload length. After the header is the payload, and usually a cyclic redundancy check (CRC) to ensure payload integrity.

A popular framing protocol is the Generic Framing Procedure (GFP) put forward by the International Telecommunications Union (ITU) [42]. The GFP standard is instructive reading, though beyond the scope of this document.

6.4.3 Modulation

After the data to be transmitted is framed and wrapped in FEC, it is ready to be converted to a form that will allow it to travel in an analog world. This process is known as modulation (and at the receiver, demodulation), and it is a concept as old as radio itself. Modulation is a means of embedding information in a radio frequency (RF) signal for broadcast [41]. Perhaps the two most common, or at least well-known, types of modulation are amplitude modulation (AM) and frequency modulation (FM) used by commercial radio stations.

Most modulation "schemes" (as they are called) have relatively unambiguous names. For example, amplitude modulation uses the data signal (e.g. voice or music) to change (modulate) the amplitude of the transmitted signal. FM uses the data to change the frequency of the transmitted signal [41].

In digital systems, the nomenclature is slightly different. In basic digital modulation schemes, there are only two states to represent (one and zero), as opposed to the infinite number of states that must be represented to send analog data. Thus, digital modulation schemes are generally referred to as "keying" schemes: one "state" of the transmitter is keyed to either a one or a zero [41]. For example, frequency-shift keying is the digital equivalent of FM. One frequency state is keyed to representing a one bit, while the other is keyed to representing a zero bit.

Digital modulation generally requires more complexity at the receiver than analog modulation. Digital waveforms are sampled by the receiver, meaning they are not perfect replications of the waveform as it was transmitted. This does not necessarily mean that information is lost, however; if the signal is sampled at twice the data rate or greater, the original signal will be able to be reproduced. This

is known as the Nyquist rate [3]. An excellent treatment of Nyquist sampling is available in [43]. There is no guarantee that the samples taken correspond to the optimal points of the signal, so some level of filtering and resampling is usually necessary; this process is known as clock recovery [41]. After this, the signal may be demodulated, and the original data recovered.

Given the complexity and estimation involved in receiving a digital signal, it is no surprise that errors may be introduced. FEC is the primary means of combatting these errors.

6.5 Forward Error-Correction Coding (FEC)

FEC sits between framing and modulation/filtering in the DSP chain (sometimes called "channel coding," see **Figure 2**). It prepares the data stream for the vagaries of the real world by allowing the receiver to cope with error and uncertainty in the received signal.

The most basic means of dealing with errors in received data is to request a retransmission of the data stream. This requires, of course, that the receiver have a means to contact the transmitter, which is not always the case. Additionally, if the receiver is experiencing errors from the channel, there is no guarantee that it will successfully contact the transmitter. Scenarios also exist where it is not practical to request a retransmission: in the example of interplanetary communications, the delay between Earth and mars is ~20 minutes [44], so a lost message could cause a delay of up to 40 minutes.

FEC seeks to mitigate the problem of message errors and reduce the need for retransmissions by adding redundancy and structure to the transmitted message, thus allowing the receiver to correct some (hopefully all) of the errors itself without needing to request a retransmission.

6.5.1 The FEC Paradigm

Hamming proposed a geometric model of message encoding [7] that both helps ground Shannon's model in a more concrete example and underpins the basic paradigm of what FEC is trying to accomplish. For this example, let the set of all three-digit binary messages represent the coordinates of a cube with sides of unit length on a Cartesian plane. Consider the following scenarios:

1. If any set of three bits may be transmitted, all eight vertices of the cube represent valid messages. If any bit in the transmitted message is changed, the result is the coordinates of a vertex of the cube, which is by definition a valid message word. The receiver can never be sure that there were no changes to the message (high uncertainty), but as much information as possible was transmitted.
2. If instead there exists a mapping from the set of two-bit messages {00, 01, 10, 11} to a set of four three-bit messages {001, 010, 100, 111}, there will exist no valid messages that lie on adjacent vertices. It the receiver receives (for example), 101, it will recognize this as an error. Assuming that errors are randomly distributed throughout the message, there is no way to determine if the intended message was 001, 100, or 111. In this example (paraphrased from [7]), the receiver has less uncertainty about the received message, but less information (only two bits) was received in the same period as Scenario 1.
3. Scenario 2 can be extended to the mapping of all single-bit messages {0, 1} to the set of three-bit messages {000, 111}. In this scenario, there are no valid messages whose adjacent vertices are also adjacent to a valid message. Assuming again a random probability of errors at each message bit, the receiver can have confidence that any message with only a single "1" bit was likely an attempt to transmit a zero bit, and vice versa. This receiver has the lowest uncertainty of any scenario, but data is received at only one third the maximum rate (see "The Repetition Code," in **Section 6.5.4.1** below).

The above scenarios demonstrate the logical basis for both the Hamming Distance (see **Section 6.5.2**, below) and Hamming Notation (see **Section 6.5.2**, below). They also reveal two underlying assumptions of the Shannon definition of information: it applies only when information is being transferred, and only when this transfer may fail. The possibility that a transfer of information may fail is the underpinning of the Shannon-Hartley Theorem [4] and causes certainty and information rate to

become linked quantities, similar to precision and recall [45], or the Gabor Limit (a treatment of which is available in [43]).

6.5.2 Hamming Notation

Many of the properties and metrics of FEC codes were first posited in [7]. As part of his work, Hamming created the first set of vocabulary for describing FEC codes in a general fashion at a high level. A FEC code is generally said to have the following properties:

- A message size *m* of user data to be transmitted
- A codeword size *n* of total bits transmitted
- A number of bits $k = n - m$ used for error detection and correction
- A redundancy ratio $R = n/m$ indicating the amount of FEC overhead
- An effective data rate of $R = m/n$

Note that effective data rate was not addressed by Hamming in his original paper and that some references may use *k* as the message size instead of *m*. Note also that both redundancy ratio and effective rate use the same letter for representation despite being inverses of each other. The user is cautioned to check the nomenclature used by the source carefully.

Hamming also created a generalized notation for FEC codes that captured to essential metrics of the code into an easily decipherable format. In Hamming's original nomenclature, codes would be referred to as an (*n,m*) code. For example, the classic Hamming-(7,4) code discussed below maps four message bits to a total of 7 transmitted bits.

After the work of Golay on non-binary codes [12], the nomenclature for FEC codes was extended to $[n,m,d]_q$ where "*q*" is the alphabet size. In this nomenclature, the Hamming-(7,4) code is a $[7,4,3]_2$ code, as it deals with binary data where the alphabet size is two (see below). Note again that some sources refer to this an a $[n,k,d]_q$ code where *k* is the message length.

The value “d” of a code is also known as the Hamming Distance. Hamming Distance is defined as the number of bit positions that differ between two codewords. For instance, the codewords “000” and “111” have a Hamming Distance of 3 from each other. Every FEC code has a minimum Hamming distance between all of its valid codewords that allows it to perform error correction (see **Section 6.5.3**)

6.5.3 The Principle of Error Correction

The core principle of FEC is that the recipient of a stream of information should be able to use their knowledge of the FEC code to detect and correct errors in the information received [37]. The process of FEC encoding may be described as embedding the message of interest into a larger pattern already known to the receiver to create more “space” between messages (refer back to the examples in **Section 6.5.1**).

Many digital channel models assumption that (1) errors are equally likely to occur anywhere in a message, (2) the probability of the transition 1 → 0 is equal to the probability of the transition 0 → 1, and (3) errors are randomly distributed throughout the message. This model is known as a binary symmetric channel [37]. In practice the first assumption often fails, as burst errors, errors that affect several sequential data symbols [8], may be introduced by a transient event. Examples include a lightning strike near a cable line or a cellphone user in a car passing behind a stand of trees. Burst errors may be modeled as a Markov process (a transition from a “good” state to a “bad” state and back) [46]. The relative chances of each transition are controlled by the quality of the channel: it may be mostly good (low probability of a good --> bad transition, high probability of the reverse), or mostly bad (with the opposite holding true). If the channel is mostly good, burst errors may themselves be rare. If this is the case, they can be compensated for.

Interleaving, whereby data bits are moved around in a block of data according to a pseudorandom pattern, can help to mitigate the effects of a burst error [47] [48]. Since the data are randomized before transmission, a burst error will corrupt several adjacent symbols of randomized data.

Undoing the interleaving process at the receiver will cause the formerly adjacent errors to become (pseudo)randomly distributed throughout the data block. Note that for this to be effective, multiple codewords of length *n* must be interleaved together or the errors in the block will just be rearranged.

Assuming that errors are randomly distributed in a block of received data, the FEC code must allow the receiver to recover the original message while correcting any errors introduced by the channel. The number of errors a code can correct or detect is determined entirely by the Hamming Distance between codewords. The capabilities of a code may be determined by the following equations:

- A code can detect a number of errors not greater than d/2, rounded down.
- A code can correct a number of errors strictly less than d/2.

Note that from these equations a code with an odd distance between codewords can correct every error it can detect, while a code with an even distance between codewords can detect one more error than it can correct.

6.5.4 A survey of Common Error-Correction Codes

The history of FEC is long and filled with the work of many brilliant minds. As such there are dozens of error-correcting codes in existence, all with different purposes, capabilities, and mathematical underpinnings. Discussing and implementing all, or even most, FEC codes is well beyond the scope of this work. Here, only FEC codes with either historic significance or current applications that are amenable to implementation in the Julia programming language are reviewed and implemented.

6.5.4.1 The Repetition Code

Prior to the work of Hamming, error correction was handled by simple majority logic decoding, such as the "2 out of 5 codes" used in the Bell Relay Computer [49] or the "3 out of 7 codes" used in radio telegraphy [50]. The message was simply broadcast multiple times (usually an odd number) and for each message bit, the value that appeared most often was taken as the correct value. Note that in

the case of telegraphy, these values are “on” and “off,” while for digital systems they are “0” and “1.” In both cases the outcome is the same.

In Hamming’s notation (see **Section 6.5.2**, above), these codes are $[5,1,5]_2$ and $[7,1,7]_2$ codes, respectively. While they can detect and correct a large number of errors, they incur a great deal of overhead versus their data rate. Their efficacy is also largely based on how they are implemented: if the code sends each value n times in a row, a burst error of *(n/2)+1* will cause a symbol to be decoded incorrectly, whereas repeating the message *n* times in a row will not have this issue.

It is also possible to create repetition codes where the message is repeated an even number of times. For example, a six-repetition code ($[6,1,6]_2$) is possible. Compared to the $[5,1,5]_2$ code, this code can correct the same number of errors. It can detect up to three errors, but there would be no way to determine whether “000111” was intended to be all zeros or all ones. Even codes can thus correct as many errors as an odd code with one fewer repetitions but detect one more for the cost of additional overhead. Whether this trade-off is beneficial depends on the application, though in all use cases presented above an odd number of repetitions is used.

The primary benefit of repetition codes is that they are easy and inexpensive to implement. Support for repetition codes of arbitrary size has been created for the Julia programming language in part of this project.

6.5.4.2 The Parity Bit

The sum of the number of “1” bits in a set of bits (generally a byte) is often referred to as the Hamming Weight [51]. “Hamming Weight” more generally refers to the number of elements in a set that differ from the zero element of that set, or equivalently the Hamming Distance of a set from the set of zeros [51]. When restricted to binary, the Hamming Weight modulo 2 essentially indicates whether an odd or even number of bits are set to one.

If data messages are randomly selected from the set of all possible messages, there is no way to know *a priori* whether the received message originally had an odd or even Hamming Weight. The addition of a parity bit solves this problem.

The concept of a parity bit is simple: the system designer selects either an "even" or "odd" Hamming Weight (known as "parity" in this context) for every transmitted codeword. If the parity of the message is not equal to the desired parity, a "1" bit is added to the message to form the codeword, otherwise a "0" bit is added.

If an error is introduced during transmission of the data, the calculated parity of the received codeword will not match the desired parity, and an error is known to have occurred. A parity bit can detect any odd number of errors, but any even number of errors is invisible as the parity will not change.

Using a parity bit provides no facility for error correction, but it allows some level of error detection with low overhead that can be implemented with a single XOR gate in hardware. Support for parity bit error checks of arbitrary size has been created for the Julia programming language in support of this project.

6.5.4.3 The Hamming Code

The Hamming Code, originally used to correct errors in computer programming punch cards [10], was first put forward by Richard Hamming in 1950 [7]. Since the computers at Bell Labs could use parity bits to detect errors in an input, Hamming suspected that there may be a way to structure the input so that errors could also be corrected without outside input. These codes were the first example of "linear codes," that is, error correction codes that function based on linear algebra principles, and where any linear combination of valid codewords is also a valid codeword.

Linear codes usually employ a "generator matrix" of binary data that is combined with a message word on the encoding side to produce a codeword. The message is part of a larger stream of binary data that is treated as a vector of a fixed size. This allows the message word and the generator

matrix to be combined using any linear algebra operations (usually a dot product). A corresponding "parity check matrix" on the decoding side is used to detect any introduced errors (again, this generally involves a dot product). The process of transforming codewords back into message words varies greatly depending on the code. The matrices used in encoding and decoding are often specified as "G" and "H" respectively.

The Hamming code uses the minimum number of parity bits to accompany the data transmitted such that each data and parity bit is checked by a unique combination of parity bits. In this regard they are called "perfect codes;" [12] they could not possibly contain fewer non-message bits and retain their error correction ability.

The most basic, and best-studied, Hamming code is the "Hamming-(7,4)" code, which is a $[7,4,3]_2$ code, with three parity bits and four data bits. The parity bits are labeled 1-3 and distributed throughout the final codeword at locations $2^{(n-1)}$. The data bits are filled into the remaining four slots of the codeword in order. The resulting codeword has the form [PPDPDDD], where *P* represents a parity bit and *D* represents a data bit.

Each parity bit n protects entries in the codeword where the n^{th} bit is set. In the case of the Hamming-(7,4) code, this means that the first parity bit protects itself and the first, second, and fourth data bits. The second parity bit protects itself and the first, third, and fourth data bits, while the third parity bit protects itself and the last three data bits. This pattern can be extended to codes of arbitrary size. A visual representation can be seen in **Figure 3**, below.

Bit Number	1	2	3	4	5	6	7
TX Bit	P1	P2	D1	P3	D2	D3	D4
P1	■		■		■		■
P2		■	■			■	■
P3				■	■	■	■

Figure 3: A visual representation of parity protection in a Hamming Code. *Content derived from* [7].

The Hamming generator and parity check matrices are derived from the parity bits. Removing the parity columns from **Figure 3** produces a matrix that maps the parity bits to the data bits they protect. This matrix is known as "A" in the Hamming code nomenclature, and it is the basis of the entire code. The generator matrix is a permutation of the augmented matrix created from $(I_k|A^T)$. Each codeword is the dot product of G^T and the data word. Encoding has a complexity of $O(n^2)$.

The parity check matrix is like the generator matrix, being $(A|I_{(n-k)})$, permuted to look like **Figure 3**. If the received codeword "c" has no errors, H * c will be the zero vector. If there is an error, the result will be a vector indicating the location of the error, called the *syndrome*. The error can be corrected by reading the syndrome as a bit index and flipping that bit in the received codeword. Parity check has a complexity of $O(n^2)$ as well.

Codewords are decoded by multiplication with a recovery matrix $R = (I_{(n-k,k)}|0_{(n-k)})$. R is permuted so that each column from the identity matrix corresponds to a data column of the received codeword. The dot product R * c regenerates the original data word. Again, decoding has a complexity of $O(n^2)$.

Hamming codes are simple and reasonably efficient, but to achieve high efficiency the data block size must be increased, and so must the number of parity bits. Each block remains capable of correcting only one error, however, so in many cases a "stronger" code may be needed.

Hamming codes are a natural fit for the Julia programming language due to its built-in support for most linear algebra operations. Support for Hamming codes of arbitrary size has been created for the Julia language in support of this project.

6.5.4.4 Hadamard Codes

The Hadamard code is one of the most-studied codes in coding theory, mathematics, and computer science. Hadamard codes are locally-decodable linear block codes that may be constructed in several different ways. Also known as Walsh codes, or Walsh-Hadamard codes, they were used as the FEC scheme on the NASA Mariner 9 probe.

Hadamard codes map a message of length *m* to a codework of length 2^m. The distance between all codewords in 2^{m-1}, making the Hadamard code a $[2^m, m, 2^{m-1}]_2$ code. They are defined as the set of all dot products of the message word with the generator matrix modulo 2, where the columns of the generator matrix are the set of all possible message words in lexicographical order [52]. Dell and van Melkebeek note that Hadamard codes may correct a great number of errors but are generally not useful in practice because of their low data rate relative to overhead [53].

There is no polynomial-time algorithm for decoding Hadamard codes deterministically [53]. Instead, Hadamard codes are generally decoded probabilistically using "oracle decoding." Oracle decoding allows for local decoding by examining only a small number of message bits and iterating multiple times to boost confidence in the correctness of the decoding.

Support for Hadamard codes with a naïve decoder was created in Julia for this project. As no reference implementation was found, these codes were not explored as extensively as other codes presented here.

6.5.4.5 Golay Codes

Hamming codes are sometimes referred to as "perfect codes," that is, there are no codes with the same message word and codeword sizes that can correct more errors, and correspondingly there are no smaller codewords that can correct the same number of errors as a Hamming code in message words of that size [54] [55]. Perfect codes are said to observe the Hamming Bound, which is defined as:

$$A_q(n,d) \leq \frac{q^n}{\sum_{k=0}^{t} \binom{n}{k}(q-1)^k}$$

Figure 4: The mathematical representation of the Hamming Bound. *Image is public domain via Wikipedia.*

The Hamming Bound defines the number of codewords of size *n* and Hamming Distance *d* that can exist for an alphabet of size *q* and a message size of *k*. A code is perfect if and only if the number of codewords is equal to the maximum possible number. The Hamming Bound arises cleanly from visualizing codewords as spheres of radius *t=(d-1)/2* (the maximum number of errors a code can

correct), and trying to "pack" as many as possible into the space provided by the constraints of the code size [55].

Shortly after the publication of the Hamming Code, Marcel Golay discovered a new perfect $[23,12,7]_2$ and $[11,6,5]_3$ code, as well as extended versions of each with additional parity [12]. The perfect Golay-(23,12) code is the only known code that can correct 3 errors in block of 23 elements [54]. The Golay code was used on the Voyager 1 and 2 spacecrafts and was specified for use in automatic link establishment by MIL-STD-188-141B [56].

Like the Hamming code, the Golay code is a linear block code that maps a message word of 12 bits to a codeword of 23 or 24 bits by multiplying it with the transpose of a generator matrix G. The basis for G for the perfect code was derived by Golay from Pascal's Triangle, while the basis of the extended code G can be viewed and generated in several ways, notably by viewing the codewords as a linear subspace of all possible codewords where each codeword differs by eight points (including the zero codeword). The full generator matrix is the augmented matrix $[I_m|G]$. Note that the generator matrix for the extended Golay code is symmetric, which makes the parity check matrix (H) equal to G^T [57] (since G is transposed for encoding H is simply the generator matrix before transposition).

The Golay decoding algorithm in the absence of errors is quite simple: since the first *m* rows of the generator matrix are the identity matrix, the message word is the first *m* bits of the received codeword. The decoding algorithm in the presence of errors can grow quite complex, however, and is not practical to reprint in entirety here. A thorough treatment of the process of decoding is available in [57] and was used to implement the Golay decoder in this project.

Since the Golay's initial paper was once called the "best single published page" in coding theory by Berlekamp [58] (of Berlekamp-Massey and Berlekamp-Welch algorithm fame, see Reed-Solomon Codes, below), a Julia implementation was created in support of this project.

6.5.4.6 Reed-Solomon Codes

Only a few years after the work of Hamming, Shannon, and Golay, the first Reed-Solomon codes were posited by Irving Reed and Gustave Solomon [59]. These Reed-Solomon codes were much more complex than Hamming and even Golay codes but were correspondingly much more powerful.

The fundamental idea behind Reed-Solomon codes is that a stream of message words of some size could be viewed as either the values or coefficients of polynomials over a finite field. By treating the message as a finite field, structure is automatically imposed without any additional work. This allows for certain mathematical manipulation of the data that would not otherwise be possible.

Codewords in a Reed-Solomon (RS) code are composed of some number of message words of size *m* each composed into blocks of size 2^m-1 including message words and parity words. It is common in digital systems to use m=8, since that causes each word to be 8 bits (1 byte) long. The codeword block is thus 255 bytes. Many standards, including DVB-S, place 223 message words and 32 parity words into each codeword block [60]. In Hamming notation, the DVB-S RS code is a $[255,223,32]_8$ code. The DVB-S RS code can correct up to 16 errors, or 32 erasures if their location is known ahead of time.

The RS code is considered optimal in that it approaches the Singleton bound [61], though obviously it could not have been designed to this bound as the code was developed four years prior to the publication of Singleton's paper. Codes at the Singleton bound are called "maximum distance seperable," or MDS, codes; they have the maximum number of message words that can be protected by the prescribed number of parity words without reducing the Hamming distance between any two messages.

Since RS codes work on message words of sizes greater than one bit (usually 8 bits), this provides them with very different error tolerance and correction properties compared to any other type of FEC reviewed here. Since RS codes treat each message word as a single value, any number of errors can be corrected *if they occur in the same word* [59]. To the RS code, a message word with eight errors is as easy to correct as one with one error. This property makes RS codes particularly tolerant to burst

errors, as they treat a burst error of up to *m* as a single error to correct. It is why RS codes are so popular for storage media since storage media is likely to suffer specific physical damage to part of its data and is less likely to be randomly damaged. Interestingly, it also makes RS codes weak in the face of random bit errors since each bit error will corrupt an entire message word. For this reason, it is popular to concatenate RS codes with convolutional codes so that one can compensate for the other's weaknesses.

RS codes are an entire family of codes, which includes cyclic, linear, and non-linear codes. This flexibility allows for many different methods of encoding and decoding, though only two are covered below: Reed and Solomon's initial specification, and the one used in this project.

In Reed and Solomon's original paper [59], to encode a message the message words are mapped to a polynomial that is evaluated at several different points. The code is essentially a linear code where the generator matrix is the transpose of a Vandermonde matrix and the output codeword is the dot product of that matrix and the message word. Reed and Solomon proposed a theoretical method for decoding the codeword by testing different encoding polynomials against it, but this decoder was never implemented as it is computationally infeasible for all practical codes.

Another, more common means of encoding RS codes is to treat the message words as the coefficients of a polynomial. This polynomial is divided by an implementation-defined generator polynomial to produce a remainder, which is appended to the message words [62]. One of the side benefits of this method of encoding is that the original message words are part of the transmitted codeword. In resource-constrained systems where errors are not expected, the decoding process can be skipped.

There have been many RS decoders over the years, with one of the most popular being the Berlekamp-Massey algorithm [63] [64]. Since the transmitted RS codeword is a polynomial divisible by the generator polynomial, if the received codeword is not divisible by the generator polynomial, errors

must have occurred. These errors can be modeled as a polynomial of unknown order and coefficients that has been added to the transmitted codeword. If the error polynomial is calculated, it can be subtracted from the received message to regenerate the transmitted message.

The Berlekamp-Massey algorithm is an iterative algorithm that seeks to find the minimum number of errors (and their locations) such that the syndromes (essentially errors) are zero. The algorithm terminates when increasing the number of assumed errors does not change the error locator polynomial [63].

Since RS codes are still widely used today, for example in 802.11n [16] and DVB-S/DVB-T [60], it is worth evaluating the ability of the Julia language to implement RS codes efficiently. Support for RS codes using the Berlekamp-Massey decoder was created for the Julia language as part of this project.

6.5.4.7 Convolutional Codes

The codes described thus far are from the class of "block codes," that is, they may be conceptualized as a block of data followed by (or intermixed with) parity bits that protect that block of data. Convolutional codes are instead stream-based: they may be pictured as a "sliding window" through which the data stream travels, generating output.

Convolutional codes were developed by Peter Elias of MIT in 1955 [65]. Convolutional coding involves convolving at least two known polynomials (or more accurately, two sets of bits interpreted as the coefficient representation of polynomials) with an input bitstream, summing the convolution modulo 2, and outputting the results. Note that only the parity bits are transmitted, not the data bits. After each convolution, the input bitstream is advanced one bit and the process is repeated. Convolutional coding is simple to implement as a shift register. An example of convolutional coding is shown in **Figure 5**.

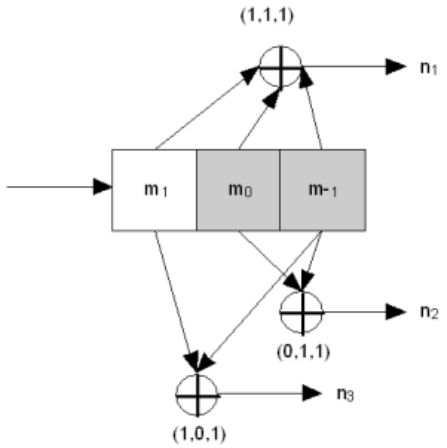

Figure 5: An example of generating parity bits for a rate-1/3 convolutional code with constraint length 3. *Image is in the public domain.*

Convolutional codes are generally defined by two parameters: their rate and constraint length. Each may be changed independently of the other with varying effects on the performance of the resulting code.

Since the data bitstream is convolved with every polynomial to generate a parity bit, the rate is inversely proportional to the number of polynomials used to encode the data [37]. For example, a rate-1/2 convolutional code would use two polynomials, while a rate-1/4 would use four. More polynomials mean more parity bits and thus a better error correction capability, but also more overhead in the transmission. To revisit Shannon's definition of information, adding additional polynomials decreases the amount of new information gained by the receiver in a message, thus reducing information but making it easier to receive. The choice of code rate is largely determined by the requirements of the end application: DVB-S (satellite TV) uses a rate-1/2 encoder [60], while deep-space missions generally use more complex codes [66].

The "constraint length" of a convolutional code is equal to the number of bits in each polynomial in binary coefficient form. That is, the constraint length is equal to the order of the convolutional polynomials plus one [37]. Higher constraint lengths allow for better error correction, but at a cost of higher decoding complexity: the Viterbi Algorithm (used for decoding, see below), is exponential WRT constraint length. DVB-S and the Voyager satellite program used a constraint length of 7, while most mars rovers use a constraint length of 15 [66].

Unlike block codes, where both the parity and data bits are transferred, convolutional codes transmit only the parity bits. They can be modeled as a state machine [37] that emits parity bits based

on the input data stream. The state machine model is used in the most popular convolutional decoder, the Viterbi Algorithm.

Developed by Andrew Viterbi in 1967 [67], the Viterbi algorithm is a dynamic programming algorithm that determines, based on a set of outcomes, the most likely series of states passed through to create them. The Viterbi algorithm functions for any Markov data source and is often used in speech-to-text applications [68]. In the context of convolutional codes, the Viterbi algorithm is used on the receiver to decode the received bitstream.

Since the receiver receives only the parity bits resulting from the convolution of the data stream with the FEC polynomials, it must work backwards to determine the "states" that produced the perceived "outcome." To manage complexity, the Viterbi algorithm uses a time-invariant trellis of fixed size to track the states. The "cost" in errors of each transition is tallied, and the receiver returns the data bitstream with the lowest number of errors in the path. An example of a Viterbi Trellis is shown in **Figure 6**.

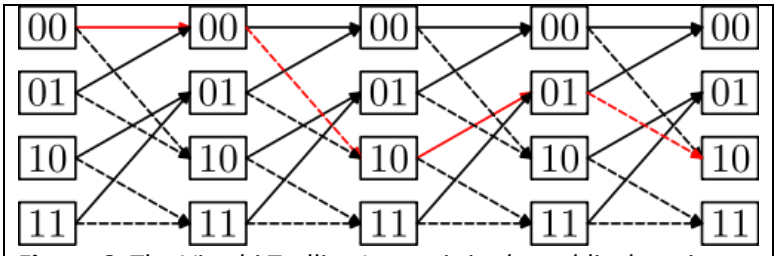

Figure 6: The Viterbi Trellis. *Image is in the public domain*.

Because convolutional codes are so powerful, it may be desirable to reduce their error-correction capacity and increase their data rate. Since a minimum of two polynomials must be used, the only way to increase the rate of data transfer with a convolutional code is to use a technique known as puncturing [69]. In a punctured convolutional code, a predefined set of parity bits are discarded at the transmitter after encoding [70]. Standard puncturing matrices are widely available [60] [70]. The net effect on the transmission is that the same message may be transmitted with lower overhead, but fewer errors can be corrected because of the reduction in parity bits.

An example parity matrix for a rate-½ code may be p = {101, 110}. In this case, out of every three message bits, six parity bits (two polynomials times 3 bits) are generated. Using the puncturing matrix shown above, the second bit from the first polynomial and the third bit from the second polynomial are discarded, and a total of four bits are sent. Since there were two polynomials, it takes 3 data bits to generate six parity bits. This means that the final data rate is ¾.

The receiver of a punctured convolutional code inserts zeros into the received bitstream in accordance with the puncturing matrix. Sometimes this will result in an error being introduced to the data stream (i.e. a 1 was discarded at the transmitter, but a zero was inserted at the receiver), which will have to be corrected by the decoder. Since errors are introduced by the encoding, fewer "real" errors will be able to be corrected. This is the cost of the higher data rate and is referred to as reducing the "free distance" of the code.

6.5.4.8 Concatenated Codes

There is no limitation on the number and type of FEC codes that may be employed on a message, not the order in which them are applied. In fact, it is common to pair a Reed-Solomon code with a punctured convolutional code [60] [66]. As noted in **Section 6.5.4.7**, above, a punctured convolutional code allows higher data rates than a traditional convolutional code at the cost of less error-correction capability. Adding a Reed-Solomon code to the mix incurs some level of additional overhead (which blunts the advantage of using a punctured code) but allows for more error correction than a non-punctured convolutional code alone.

Concatenated codes are not without their issues. In addition to increased overhead, it has noted by [66] that any mistakes made by the "outer" decoder are passed on to the "inner" decoder, even if they are not mistakes that the inner decoder would have made itself. One defense against this problem is to express the output of the outer decoder as a probability (a "soft" decision) instead of an

absolute answer (a “hard” decision). This allows the inner decoder to make use of the soft decision values in what is called a “maximum *a posteriori* decoder,” or MAP.

It follows that a single algorithm should be able to work from its own soft decision information to arrive at the best possible answer iteratively. Most modern FEC, including Turbo codes and Low-Density Parity Check codes, follows this reasoning.

6.5.4.9 Modern FEC: Turbo and LDPC Codes

Many modern communications standards use convolutional and RS codes as their basis, including DVB-S [60] and 802.11n [16]. As the demand for higher data rates in contested spectrum grew, however, new codes were needed that could approach the Shannon limit of a channel while maintaining high error correction capability. Thus, Turbo codes and Low-Density Parity Check (LDPC) codes were developed and optimized.

“New” in the case of FEC is a relative term: Turbo codes (from the French: Turbocode) were developed in 1993 [71], while LDPC codes were first developed in the 1960s [72] but not realized until 1996 [73]. These codes have found there way into several modern communications standards, including DVB-S2 and 802.11ac.

Both types of codes are most notable for their decoders, which use an iterative belief-propagation algorithm that may operate any number of times to increase its error correction ability. Unfortunately, both Turbo codes and LDPC codes are known to be extremely computation-intensive and there are scant reference implementations available. One promising implementation (github.com/blegal/Fast_LDPC_decoder_for_x86) required non-standard C++ compiler extensions only available with the Intel C++ compiler.

Due to time constraints and the absence of any significant reference implementation, neither Turbo nor LDPC codes were implemented as part of this project.

6.6 Implementing Communications Systems in Software

As mentioned previously, software has a much faster adoption and upgrade cycle than hardware. With the rapid pace of improvement and proliferation of communications systems, software can appear as an attractive alternative to hardware for implementing these systems. Communications systems generally require real-time performance, however, which can be daunting to achieve on general-purpose, nondeterministic platforms.

To achieve the required speed, low-level programming languages are often used, which are generally considered less-than-friendly towards the developer. On the other end of the spectrum are "quick and easy" languages like Python, which promise users to ability to implement software communications systems rapidly at the cost of execution speed. In a real-time system, may prove to be an untenable tradeoff. Systems have attempted to bridge the gap between the two types of languages using tools like SWIG [29], with varying results. Recently, the Julia programming language was developed to address this need and seeks to bridge these two worlds and provide "a way to implement fast code, fast."

6.6.1 Python and Other 4th-Generation Languages for DSP

The benefits of scripting languages over lower-level languages are well-documented in the literature. In 1998 John K. Ousterhout argued forcefully in IEEE Computer that scripting languages could reduce program construction times by a factor of 5-10 [28], a conclusion which is borne out by research summarized by Steve McConnell in *Code Complete II* [27]. Since programmers tend to write the same number of lines of code per year regardless of programming language used [74], programming languages that express a greater number of instructions per line of code allows programmers to "say what needs to be said" in less time. Python (and other scripting languages such as Tcl) allow many more instructions per line, and thus higher productivity, than languages such as C/C++ and FORTRAN [75] [76].

The reduced number of lines of code required to implement the project in a higher-level language has additional effects on project schedule. Available data for Visual Basic (VB) and Delphi

versus C/C++ indicate that a switch to higher-level languages can save 75% on coding effort [76] and design time [77], but no effect was seen on architecture and testing. It has been noted be McConnell [78] that such a reduction in effort will lead to an ~25% decrease in project schedule overall. Python is estimated to be even more productive than VB [28] [27], so the difference may be greater. Of course, the productivity of "third-generation languages" such as C++ has likely improved in the intervening twenty years as well.

Reduced code size also aids the programmer in achieving maintainability. The Software Improvement Group (SIG) uses as one of its maintainability standards the length of functional "units" of code (generally a function), recommending a unit size of 15 non-empty, non-comment lines or fewer [79]. The head of the SIG also notes in Building Maintainable Software that software quality tends to decrease in proportion to the size of the project codebase [80] (this data is largely summarized from the works of Capers Jones, whose work is also the basis for many conclusions in [27] and [78]). Research by Stefnik and Seibert [81] into the readability of programming languages indicated that "traditional" C-style syntax, as seen in C/C++ and Java, proved to be a significant barrier to novice programmers. In fact, Java proved no more intuitive to the subjects of the study than a language wherein the keywords had been randomly selected from a list. Previous work [82] found that half of students in the top quintile of their Java class submitted code with syntax errors, with this number rising to 73% for the bottom quintile. Python (as well as Ruby and Quorum) proved more intuitive to novice programmers.

For all their advantages, high-level languages have limitations as well. The best-known limitation of high-level languages is speed: even in his 1998 paper arguing the merits of scripting languages, Ousterhout acknowledges that they impose a 10-to-20-fold decrease in execution speed over compiled languages [28]. Even as recently as 2013, it was argued that improvements in scripting language speed could largely be attributed to hardware improvements, especially in resource-

constrained environments [83]. More recently, Python 3 performed poorly relative to C/C++ in both the Julia language micro-benchmarks and the "Computer Language Benchmarks Game" [84].

Since one of the Julia programming language's "claims to fame" is its speed relative to Python for the same implementation complexity, a not insignificant amount of digital ink has been spilled both in support of and in opposition to the claims of Python's speed. Work by members of NASA's "Modeling Guru" community explored the speed of algorithm implementations in various programming languages on several occasions [85] [86]. In most cases, they demonstrated Python code operating at speeds within a factor of two of Julia. One could argue that many of the "fast" Python implementations used additional libraries or packages (Numba, Numpy) and that the pure Python implementations were generally slow with respect to Julia, but that is not always the case. Similar work has been done by developers at IBM [87], showing that in many cases even the core Python library has options that allow it to rival Julia in speed, and that there are a cadre of tools available for programmers looking to optimize Python code already available (the effect of community and tool support on programming languages is discussed below).

It should be noted, however, that the applicability of computer benchmarks to real-world performance has been disputed [88] [89]. To extrapolate from [88], software benchmark scores may be a side-effect of the Peopleware adage "What gets measured, gets managed [90]."

As a final note, some high-level languages (Python included) also do not include compile-time safety checks or type safety, which can limit their applicability to large systems. Many scripting languages (including Tcl and AWK) rely on the string as the primary data type [28], which is partially the genesis of the pejorative slang jargon "stringly-typed program [91]." For its part, Python generally delays type-checking until just before an AttributeError takes down the whole program.

Python relies heavily on "duck typing," a type system wherein type checking is delayed until runtime, and the actual type of an object is less important than what can be done with it [92]. The

phrase “duck typing” comes from the colloquialism “If it looks like a duck and quacks like a duck, it’s probably a duck.” The idea, in terms of programming, is that since both the list data type and the string data type support random element access via operator [] with an integer index, they can both implicitly be valid arguments to a function that utilizes only this functionality. Issues may arise from what Joel Spolsky of Stack Overflow calls the “Law of Leaky Abstractions:” every abstraction has instances wherein it fails to conform to the desired behavior [93]. Consider the following Python code fragments:

```
# Legal in Python 3.6.3 Anaconda

>>> salutations = { 'hello' : 5, 'hi' : 2, 'hey' : 3 }
>>> keys = data.keys()
>>> sorted_keys = sorted( keys )
```

```
# Illegal in Python 3.6.3 Anaconda

>>> salutations = { 'hello' : 5, 'hi' : 2, 'hey' : 3 }
>>> keys = data.keys()
>>> sorted_keys = keys.sort()
```

Figure 7: An example of the failure of a Python abstraction. The user expects the keys variable to act as a list, but there are cases where it does not.

The key distinction is that the return value of the `dict.keys()` function is not a list, but a `dict_keys` object, which has many, though not all, of the same options available as a list. To be used as a list, a new list object must be created from the `dict_keys` object. The Julia programming language seeks to avoid these types of errors by allowing explicit type annotations when a distinction is required.

6.6.2 C/C++ for DSP

In 2017, *IEEE Spectrum* ranked C and C++ as the second- and fourth-most popular programming languages in its annual survey [94]. Despite its status as a venerable language with enough public criticism to merit its own Wikipedia page, C++ is still a mainstay in modern programming.

Along with FORTRAN, C and C++ are generally considered "fast" programming languages. They often have aggressively optimizing compilers. In fact, the FORTRAN optimizing compiler has been called one of the best "algorithms" of the 20th century [95]. The relative execution speed of programs written in these languages has made them popular for real-time systems, and by extension DSP.

C/C++ perform well in both toy program benchmarks [84] and the Julia programming language micro-benchmarks [96], as well as work by the NASA Modeling Guru group [85] [86]. All SPEC2006 benchmark programs are written in C/C++ or FORTRAN, with more than ¾ being written in C/C++ [88].

Because of its execution speed, C++ is the language of choice to handle the backend DSP code of the popular open-source software-defined radio testbed gnuradio. Gnuradio relies on Python as "glue logic" and a frontend for its C++ "blocks," which are generally not meant for the end user to see nor interact with. Wrapper code for user-implemented C++ blocks is supposed to be generated "auto-magically" by the Simple Wrapper and Interface Generator (SWIG) [29]. The gnuradio tutorial on building C++ modules notes, however, that C++ should be used only if performance is the primary goal, otherwise it is simpler to use Python. This disconnect between burning programmer cycles and burning processor cycles on an implementation is one of the main areas addressed by the Julia programming language (see below).

C++ has been touted as providing up to 2x improved productivity over C [27], though other research summarized by McConnell [78] [76] and years of testing by DeMarco and Lister [90] have not borne this conclusion out. The language has been criticized as overly complex by such stalwarts of the field as Donald Knuth [97], and even its supporters occasionally refer to it as "the beast [98]."

The readability, and subsequently usability, of C++ has improved in recent years, with the 2011 standard adding a native concurrency API, better support for generics, and some level of reference-counting for pointers. The changes were so stark that they led the language's creator, Bjarne Stroustrup, to remark that "C++11 feels like a new language [99]." In keeping with its relentless

modernization, C++ has remained popular to this day, being one of the most popular languages on both Github and Stack Overflow (though whether the latter is an honor is debatable) [100]. According to the TIOBE company programming language ranks, C++ was the third most popular by search traffic, generating 6.452% of traffic regarding programming languages [101]. Note, however, that their data place C++ at only half its popularity of 15 years ago.

The combination of execution speed and popularity, especially in the DSP field, not to mention the uncountable number of lines of C++ code that have been written for DSP over the decades, combine to make it an attractive choice for DSP projects. The existence of large DSP libraries and tools developed in C++, especially for complex tasks like error correction coding (notably the fully SIMD libfec library and Liquid SDR), provide a significant advantage for any developer choosing C++ as their language of choice.

6.6.3 The Julia Programming Language

The Julia programming language is relatively new, having been developed in 2009 [30]. Python, by contrast, was first released in 1991, while C and C++ were released in 1972 and 1985, respectively. As such, Julia can draw on decades of research and experience in programming language design not available to older languages at their outset. A thorough, if perhaps slightly biased, overview of the technical aspects of the Julia language is presented in [30] with a level of bombast and reverence generally reserved for talking about Lisp.

In his 1973 paper "Hints on Programming Language Design," C.A.R Hoare notes many features that should be included in new programming languages for ease of development [102], including (but not limited to):

- "A good programing language should give assistance in expressing not only how the program is to run, but what it is intended to accomplish"
- "A good programming language will encourage and assist the programmer to write clear self-documenting code"

As a specialized language for numerical computing (or not, if all claims from [30] can be taken at face-value), Julia has a number of built-in features that allow the developer to express what they intend to accomplish in a clear and concise manner for most mathematical operations. The language has native support for vectors and matrices, and operator overloads are provided for any standard operations on these types. **Figure 7**, below, shows how similar Julia code can be to the underlying mathematical concepts:

Problem: Compute the dot product of a matrix A and a vector w.	
Mathematical Notation:	**Julia Code:**
A * w	product = A * w

Figure 8: Translating a matrix-vector dot product into Julia is both concise and intuitive.

Since code is read far more than it is written [103], writing clear and concise code that expresses the underlying ideas well is important to maintainability and extensibility. Julia has been criticized for using inconsistent function interfaces [104] and coding practices that reduce readability [105]. For example, the following code snippet is ambiguous:

```
A = Array(Bidiagonal(ones(5, 5), true))
```

Figure 9: Some Julia function interfaces are not as clear as they could be. Parts of the Julia API use Boolean parameters, which obscure the meaning of the arguments.

The code in **Figure 9** creates a bidiagonal matrix where the offset diagonal is above the center diagonal. Passing `false` as the second parameter to the `Bidiagonal` constructor would have placed the offset diagonal below the main diagonal. Neither result is clear from context; an enumerated type could have been used to improve readability. Julia has also been criticized for using exception-based code [104] which can be difficult to reason about and find errors in [106], especially in contexts where Julia is the only language to raise an exception and the circumstances are not exceptional (see the implementation of UDP in Julia GitHub Issue #5697).

Julia's decision to use 1-based, inclusive indexing, while having no technical issue, is seen as needlessly frustrating by many programmers [107]. Of course, Julia was designed with mathematics and science in mind, fields in which FORTRAN and Mathematica, both of which use 1-based indexing, are common.

Julia is widely touted as a language that can nearly match C or FORTRAN for execution speed while far exceeding Python, MATLAB and Octave. The Julia microbenchmarks [96] show that in most cases, this holds true. The veracity, or at least honesty, of the Julia benchmark suite has been debated over the years. The Github repository of Julia has had at least four issues opened on the topic in the last five years (#2412, #4462, #5128, #13042). In general, the primary complaint with the Julia benchmarks is that they do not use language features idiomatic to the languages that Julia claims to be besting. The finer details of the argument are largely academic, but the upshot is that the Julia benchmarks show only how Julia language constructs compare to those in MATLAB, R, and Octave, not what the end user would experience. As noted above and in [85] [86] [87], Julia tends to hold less of an advantage over Python in idiomatic applications. The advantage of Julia over MATLAB appears more solid [108].

The choice of programming language is more than just a preferred choice of syntax. Each language has its own community of developers, tools, libraries, and other support infrastructures that help developers solve problems with their language of choice. After all, any Turing-complete language can (hypothetically) tackle any coding task, but there aren't many advocating for website design in C++ (with some exceptions [109]).

In the early 2000s, VB developer and future Stack Overflow founder Jeff Atwood noted that while C# was widely seen as a superior language to VB.NET, both languages accomplished the same tasks using the same runtime, and tools were available to freely convert between the two of languages [110] [111]. By the end of 2004, however, VB.NET began to lag C# in tool support, with programmers jumping ship to C# in 2005 when it was realized that C# developers were paid more to wire the same

code as VB developers [112] [113]. Years later, Minsky and Weeks of Jane Street Capital noted that after switching to Ocaml as their institutional language of choice, they began to attract many talented developers with relevant skills [114], and hypothesize that the Ocaml language itself attracts such developers.

As far as community support is concerned, the Julia programming language has been growing in popularity, or at least in publicity and hype. The March 2018 TIOBE programming language report ranked Julia as the 37th most popular programming language by web search volume, accounting for 0.301% of search traffic for programming languages [101]. For comparison, C/C++ generated a combined 19.302% of traffic, while Python accounted for 5.869%. Interestingly, VB, FORTRAN, D, and Ada all ranked above Julia in the listings. Julia also broke into the top 10 programming languages on Github in 2017 [115]. The RedMonk programming language rankings for Q1 2018 place Julia in the middle of the pack for popularity on Stack Overflow, and in the second quartile of popularity on Github [100]. No matter which metrics are used, a strong case can be made that there is an active and extensive Julia community present. Since 2015, there has been a commercial company, Julia Computing, available to provide support and training for the Julia language [116].

Julia also has several fairly robust development tools available. Julia has a fully interactive "read-evaluate-print-loop" (REPL) shell available in both Windows and Linux with built-in support for Markdown comments. Support is available for the Jupyter interactive notebook program, and Julia has multiple integrated development environment (IDE) options, including Juno (junolab.org) and a plugin for the popular Atom IDE. Julia has a built-in testing API that allows the user to write simple, single-line tests for functions and objects using macros (or more complex tests if desired). The language also has native support for building rich documentation using Markdown, and an idiomatic documentation syntax has been defined. The Julia package system appears to be on par with the Python pip installer with regards to package management.

While there is no active work on using Julia for FEC, and little serious work on using it for DSP, there is certainly an active enough community to warrant the time investment for development.

7 TOOLS AND PROGRAMS USED

All programming and benchmarking was performed on an Intel NUC6i5SYK single-board computer (SBC) with the following specifications:

- **Processor:** Intel Core i5-6260U dual-core, hyperthreaded @ 1.8 GHz base frequency, 2.9 GHz with Intel Turbo Boost
- **Memory:** 8 GB of DDR3
- **Hard Drive:** 128 GB solid-state drive (SSD)

All programming was performed on the Ubuntu distribution of GNU/Linux, version 14.04.5 Long-Term Support (LTS). No packages were upgraded beyond the most recent available from the apt package manager as of April 4th, 2018.

Julia v0.6.2 was downloaded, built, and installed from source using GNU make. Development in the Julia language was done using a mix of tools. Initial prototypes were programmed in interactive notebooks in Jupyter using the IJulia package, with Jupyter being installed as part of the Anaconda distribution of Python. Minor testing and evaluation was performed in the Julia REPL shell. The final packages were assembled using the Juno plugin for the Atom IDE. The execution times of Julia programs were measured with the **"@timev"** macro.

C/C++ development was performed exclusively in the CLion IDE, version 2018-3 (student edition). Builds were managed with CMake 2.8.6 with GCC/G++ 4.8.4 as the compiler backend. Programs were compiled with the following flags: **"-Wall -Wextra -Wpedantic."** In the case of C++ programs, the **"-std=c++11"** flag was also used. No non-standard compiler extensions were enabled. Most FEC functions were used as-is from the Liquid DSP library (liquidsdr.org), which is written in C. The most recent version of Liquid DSP available on Github was cloned; at the time of writing this document, that is v1.3.1, commit 606df8f. The execution time of C/C++ programs was measured using the C++ **"chrono"** library, which accesses the underlying system clock (Linux makes a microsecond clock available to user-space programs).

8 DESIGN AND IMPLEMENTATION

As mentioned previously, Liquid DSP (liquidsdr.org), a mature C library was used as a reference implementation and mentor text for this project. Liquid DSP, stylized as liquid, is designed to be compiled quickly on embedded platforms and run effectively in resource-constrained environments. It avoids C++ to avoid overhead from virtual functions.

The documentation for liquid states that "… there is no model for passing data between structures, no generic interface for data abstraction, no customized/proprietary data types, no framework for handling memory management." This is generally true; though for FEC there is a consistent interface provided to the user regardless of the FEC algorithm used. The input to the functions `fec_encode()` and `fec_decode()` is always a stream of bytes, as is output, regardless of the specification of the underlying algorithm.

Since C has no built-in support for objects nor generic types, a great deal of effort was required to standardize the interface. The specifics of the individual algorithms and any state they may require is hidden within the `fec` opaque type. Fully standardizing the interface required including extensive pre-processor definitions and many fields in the `fec` object that are not necessary in all cases.

Based on experience working with liquid, a different approach was chosen for the interface of the Julia code. It was decided that it was less important to offer a standard interface for every algorithm than it was to offer an interface that was consistent with the specification of the original algorithm. Since Julia supports higher-order functions, FEC primitives could always be mapped to a stream of data if a more generic interface was desired.

In instances where liquid passed FEC calls out to the libfec library (github.com/quiet/libfec), a collection of highly optimized SIMD implementations of common and complex algorithms, it was decided that the Julia implementation would do the same. Not only would this allow for fair execution time comparisons, it would also showcase the foreign function call API of the Julia language.

9 RESULTS

Implementations of FEC algorithms were compared to reference implementations in C/C++ and (when available) Python. Implementations were compared on two counts: the number of non-blank, non-comment lines of code required to implement them, and their execution speed. Before presenting quantitative results, notes on the process of implementing these algorithms in Julia are presented as part of the evaluation of the applicability of the language to the topic at hand.

9.1 NOTES ON IMPLEMENTING FEC IN JULIA

9.1.1 Basic Repetition and Parity Check Codes

By their very nature, neither repetition codes nor parity check codes require a great deal of complexity, which is part of the reason they were developed decades before other FEC codes. The Julia implementation did not encounter any difficulties in implementing them, but neither were any expected. Very little idiomatic Julia code was used in their implementation.

9.1.2 Hamming Codes

As noted in **Section 6.5.4.3**, above, both the generator and parity-check matrices of Hamming codes use 1-based indexing, which makes Julia's use of 1-based indexing convenient. Swapping columns in Julia matrices is a simple operation, which made permutation of the G and H matrices to their final forms simple.

Since Hamming codes rely almost entirely on matrix-vector dot products for encoding and decoding, implementing both functions from the mathematical specification was a straightforward exercise, and the final Julia code tracked the mathematical underpinnings of the Hamming code very closely.

9.1.3 Hadamard Codes

Without a reference implementation or authoritative specification, Hadamard codes proved difficult to implement effectively. The only language-specific stumbling block encountered was that the

Hadamard generator matrix is zero-indexed by spec, which makes the 1-based indexing of Julia an unnatural fit for its implementation.

9.1.4 Golay Codes

The most common specification and implementation of Golay codes available is for the Golay-(24,12) extended code with an additional parity check bit, which is what was implemented here. Most of the Golay extended generator matrix is created from a single row of bits, shifted one column for every row of the matrix. The Julia `circshift(A, shifts)` function allowed for easy construction of the G matrix. Since the Golay-(24,12) code is symmetric, its parity-check matrix is simply the transpose of its generator matrix, allowing the H matrix to be constructed with a single line of code.

Like the Hamming code, encoding a message word with the Golay encoder requires only a dot product between the G matrix and the message word, which can be accomplished in Julia with a single line of code. The Golay decoder is complex (see **Section 6.5.4.5**, above), but the Julia implementation tracks the specification as closely as possible, which cannot be said for implementations in other programming languages.

9.1.5 Reed-Solomon Codes

Reed-Solomon is a powerful and commonly used error correction code often seen in high data rate applications, including most modern versions of the 802.11 wireless standard [16]. Having an efficient Julia implementation would be useful.

The primary stumbling block in implementing Reed-Solomon codes in Julia is that they require the use of finite field mathematics, something which Julia is oddly lacking native support for. There is one API available, Nemo (nemocas.org), which while extensive is not intuitive or well documented, and was not suitable for use with Reed-Solomon code as provided. Implementing a custom suite of finite field types and operations revealed several weaknesses of the Julia programming language.

The Julia language has a data type, `struct`, which is used to create immutable types. If a `struct` contains a mutable type within it, such as a `vector`, elements of the vector can still be changed despite the `struct` and the `vector` within being immutable. Since Julia is not object-oriented and has no real concept of "private" variables, it is possible to change values in a custom type that were supposed to be constant by design.

The mutability issue of Julia is compounded by the fact that arguments are passed by reference by default. This is the case in many modern programming languages, but many of those (Java, C#) have a means of preventing function arguments from being modified. In Julia, the `const` qualifier does not prevent the contents of mutable types from being modified, only the reference from being changed. Such a language construct is all too reminiscent of "`int *const err`" and similar bugbears of C code.

Beyond mutability issues, Julia allowed finite fields to be implemented clearly and concisely, with all common operations defined. Some difficulty was encountered implementing the `getindex` and `setindex!` functions for new types, as they are not well-documented in the Julia codebase.

Once finite fields were implemented, Reed-Solomon coding was not difficult. The Reed-Solomon encoder is only a few lines long. Implementing the encoder did reveal some odd behavior of the Julia language, namely that in certain scenarios assigning a range of data to a row of a matrix that had equal size to that matrix row would fail silently. This is not a documented bug of the Julia language, but did appear in several distinct locations within the FEC code.

The Reed-Solomon decoder proved much more difficult to implement, and a pure Julia implementation was not completed in the time allotted for this project. This failure had little to do with the Julia language itself, as Reed-Solomon decoders are very complex [59]. Additionally, the reference implementations studied did not use finite fields as language constructs *per se*, instead using integral operands with special function calls for basic operations.

To work around this shortcoming, the native ability of Julia to call C, C++, and FORTRAN code from shared libraries was exploited. It was discovered while looking through the code for Liquid DSP that the library did not offer any native Reed-Solomon coding ability, instead calling out to the "libfec" library. Considering this, four Julia functions were written to call out to the libfec shared library, providing efficient and compact Reed-Solomon encoding and decoding. Compared to SWIG, the Java-Native Interface (JNI), and C-FORTRAN interfaces, the Julia syntax for foreign function calls is (anecdotally) much cleaner and easier.

9.1.6 Convolutional Codes

Implementing a convolutional encoder in Julia was eased by the language's built-in support for scalar operations over entire arrays of data from a single function call. Convolving bit arrays and summing them modulo 2 required only a single line of code. Puncturing was likewise simple to implement, though it required the implementation of a stateful look-up table (LUT) that was passed as a parameter to the puncturing function. The design of the puncturing code could have been improved using objects.

Like the Reed-Solomon decoder, there was not time to implement a Viterbi decoder in pure Julia with any level of efficiency. Also like Reed-Solomon, Liquid DSP does not offer its own built-in facilities for convolutional coding, instead calling out to the libfec shared library. However, there was not time to implement a wrapper for a Viterbi decoder in Julia due to the complexity of the interface.

9.2 Comparison of SLOC Required in C versus Julia

One benefit for which 4th-generation languages are known is their ability to allow the implementation ideas in code with fewer executable lines in a generally more readable manner than 3rd-generation languages [27] [28] [29] [30] [74] [76] [77] [78]. As such it is informative to compare the number of lines of code required to implement each FEC algorithm in Julia versus C.

9.2.1 Basic Repetition and Parity-Check Codes

Implementing the "Basic" FEC package in Julia, which encompasses repetition coding of arbitrary size (including the ability to report uncorrectable errors to the user) and parity check bits, required 69 lines of executable code. The Liquid DSP implementation of rep-3 and rep-5 coding required 290 lines of executable code. Liquid DSP implements parity bits as part of its CRC sub-library, so it is not possible to obtain a clear measurement on how many SLOC were required for parity bit functionality. Liquid DSP also does not have the functionality to pass error information back to the end user. Julia allowed more functionality to be implemented with 59% fewer LOC.

The primary driver for the conciseness of the Julia implementation was the language's built-in support for array operations. The ability to automatically sum or add arrays element-wise is not only more concise, but also more readable, than using loops in C. C++11 and newer provide generic algorithms in the STL that allow for terse but obscure implementation of the same functionality.

9.2.2 Hamming Codes

Like repetition codes, Julia allowed Hamming codes of arbitrary size to be implemented in 148 lines of code. Implementing only the Hamming-(7,4) and Hamming-(15,11) codes in Liquid DSP required 225 lines of executable code. Liquid DSP does provide support for SECDED codes, which are derived from Hamming codes, and these are not supported in the Julia implementation. Their SLOC count was not counted in the above value for Liquid. Regarding Hamming codes alone, however, Julia again allowed more functionality to be implemented with 34% fewer LOC.

9.2.3 Hadamard Codes

Despite the alleged popularity of Hadamard codes in mathematics and computer science, no reference implementation (or implementation at all, for that matter) was found for the Hadamard code. In any case, implementing Hadamard codes of arbitrary size in Julia required 63 executable lines of code. Execution times are measured in **Section 9.3.3**, below, though there is no reference implementation for comparison.

9.2.4 Golay Codes

The extended Golay-(24,12) Code required only 68 LOC to implement in Julia, compared to 290 in C, a 76.6% reduction. Julia's built-in support for linear algebra operations and types proved vital in implementing linear block codes in terse and readable manners.

9.2.5 Reed-Solomon Codes

A native Julia implementation of the Reed-Solomon encoder required 132 lines of executable code, with an additional 207 lines required to implement the finite field objects used as their operands. Liquid DSP does not implement Reed-Solomon codes itself, instead providing a wrapper around the libfec library, which contains highly-optimized Reed-Solomon code implemented in vectorized assembly.

Liquid DSP uses 23 SLOC to initialize Reed-Solomon coding and ten lines to free allocated resources at program exit, while Julia requires three lines for each. A large part of this discrepancy comes from the design paradigm of Liquid DSP, that all FEC should have as consistent and generic of an interface as possible. The C programming language has no explicit support for generics, which makes this interface difficult to create concisely and idiomatically.

Implementing the encoder and decoder functions required a total of 59 SLOC in C versus 9 in Julia, an 85% reduction. The comparison is not exact, as the C implementation can encode multiple codewords at a time, which is not the case for the Julia implementation. Adding support for this functionality would have approximately doubled the size of the Julia implementation, though using Julia would still have effected a 70% reduction in code size versus C.

9.2.6 Convolutional Codes

Implementing a convolutional encoder in Julia required 31 lines of executable code, or 26 lines when parity calculations were outsourced to a C library (see below), compared to 168 lines in C, for a reduction of 81.5% in the nominal case. Implementing punctured coding in Julia required an additional 47 lines of executable code, compared to 258 in C, a reduction of 81.8%.

As part of an effort to optimize the convolutional encoder, a trivial C library was created to perform parity calculations on input codewords. Using this library greatly increased the throughput of the Julia code (see the next section) while also decreasing the number of SLOC required to implement the code.

There was not time to implement a Viterbi decoder in Julia, so no data is available on the efficacy of the Julia language for implementation versus C.

9.3 Comparison of Execution Times in C versus Julia

Since C/C++ are the *de facto* standard languages for SDR, especially processor-intensive DSP code like error correction, and Julia is alleged to run at speeds comparable to C, it is reasonable to compare a Julia implementation of FEC to a C implementation. The results of execution time and throughput measurements are presented below.

9.3.1 Basic Repetition Codes

Liquid DSP offers both a 3-repetition (Rep-3) and 5-repetition (Rep-5) repeat code. A simple benchmarking program for each was written in C++, requiring 37 non-blank, non-comment LOC. It was compiled as previously described using either "-O0" or "-O3" compiler flags. This benchmarking program was compared to a Julia implementation in Jupyter, which required only 11 lines of code to implement. The data for Rep-3 are as shown in **Table 1**, below.

Program	**C++ with -O0**	**C++ with -O3**	**Julia**
Throughput (Mbps)	240	516	107
Standard Dev	33	19	4.6
Coefficient of Variation	13.7%	3.70%	4.32%
Confidence Interval (Mbps)	40.9	23.7	5.75

Table 1: Throughput rates for C/C++ and Julia implementations of Rep-3 coding. Input was processed in blocks of 8 bits.

As can be seen from the table, the Julia implementation was slower than the C/C++ implementation in all cases by a factor of at least two. Versus the most highly optimized version of the C/C++ code, Julia was barely

20% the speed. The results are worse for Rep-5, with the C/C++ throughputs remaining similar but the Julia implementation dropping in throughput by ~40% (data not shown). A likely cause of this is memory management: the GC used only 5.95% of the runtime of the Rep-3 code but used 14.33% of the runtime of the Rep-5 code. Increasing the input size to 80 bits caused the throughput of the Julia implementation to decrease to an order or magnitude less than the optimized C/C++ implementation (data not shown).

9.3.2 Hamming Codes

Liquid DSP offers options for a Hamming-(7,4) and Hamming-(15,11) code, though they have different APIs. A simple benchmarking program for the Hamming-(7,4) code was written in C++, requiring 37 non-blank, non-comment LOC. It was compiled as previously described using either "-O0" or "-O3" compiler flags. This benchmarking program was compared to a Julia implementation in Jupyter, which required only eleven lines of code to implement. The data are as follows:

Program	**C++ with -O0**	**C++ with -O3**	**Julia**
Average (ms)	889	308	630
Standard Dev	16	13	23
Coefficient of Variation	1.82%	4.19%	3.68%
Confidence Interval (ms)	20	16	29

Table 2: Times for C/C++ and Julia to complete 1e6 repetitions of Hamming encoding and decoding.

A key difference in benchmarking the code was that the C API worked on packed bytes, while the Julia implementation worked on the message one bit at a time, as the original specification entailed [7]. To keep the C data byte-aligned without adding any extra empty bits at the end of the codeword, message words were processed in batches of 8. The Julia code operated one message word at a time, leading to the throughput results seen in **Table 3**.

Overall the data show a reasonable level of reproducibility, with coefficients of variation kept below 5% in all cases. Approximately 14.5% of the Julia program execution time was spent on garbage

Program	Throughput (codewords/second)
C++ with -O0	4.5e6
C++ with -O3	13e6
Julia	1.59e6

Table 3: A comparison of the throughput of the C/C++ and Julia Hamming Code Implementations.

collection (GC), according to the output of the "@timev" macro. C++ does not use GC, but the results of allocating and deallocating memory via RAII are included in the measurements.

An attempt was made to use the `encode_stream()` function of the Julia FEC library to match the number of codewords handled at once by the C++ program, but for large datasets this function proved to be extremely memory-intensive, requiring over 16 GiB to be allocated over 1e6 iterations. Running the test code took over 30 seconds, indicating obvious inefficiencies.

Implementing the C++ code was significantly more work than Julia. This was due to four primary factors: (1) there is no simple way to create a vector of random numbers in C++, (2) the encoding and decoding functions were written using output parameters (which is idiomatic to C) requiring that memory for the codeword and decoded message be allocated separately, (3) that memory had to be manually released after use, and (4) the Liquid DSP interface requires that the message be entered as a stream of bytes, not bits (a message size was selected to avoid wasted bits).

With respect to execution speed, C++ varied widely by optimization level, with the highest level of compiler optimization being over three times as fast as the lowest level. The level of optimization introduced by the compiler may be a consequence of the fact that C/C++ has been used as the language of choice for processor-intensive tasks for decades. The amount of time and effort spent optimizing C/C++ compilers is likely huge.

In both cases, however, the C/C++ implementation was faster than the Julia implementation ($p<0.001$). Under the best of circumstances, the Julia code was approximately 1/3 the speed of the C++ code. In the worst case, the difference was nearly an order of magnitude. Part of the issue could be due to the APIs: the C API used in the C++ code took message words in byte-wise, while the Julia code

worked bitwise, which could have caused excess memory allocation. Excess memory allocation was not entirely responsible for the results seen, as removing the allocations from the task loop reduced the runtime of the Julia code by ~25% and the C++ code by ~60%. If memory usage was not the issue, then either the underlying algorithms are very different, or Julia is not well suited for DSP.

A summary examination of the C library code revealed the former to be the case. Liquid DSP uses a LUT internally to generate the codewords, before packing them into the output array. To decode, a larger LUT is used to cover all valid and invalid codewords for automatic error correction. This will of course be faster than the three $O(n^2)$ operations required to encode and decode the message in the Julia code. The trade-off is in readability, as the C code is highly optimized but does not look in any way like the mathematical concepts it implements. The Julia code follows the mathematics quite closely but is not as fast as a result.

To explore the speed of Julia code with similar algorithmic optimizations to the Liquid DSP code, a version of the Hamming encoder was created that used the same LUT and a similar bit-packing method to the C code. Only encoding was tested for this new implementation. Using this benchmark code, the Julia encoder was as half as fast as the C encoder with -O0 optimizations, and 1/3 as fast as the C encoder with full optimization. The Julia code did not perform as well as expected, but it is at least of the same order as the C/C++ code in this case. This is close to the expected behavior based on [85] [86] [96].

Unfortunately, to achieve this level of speed the Julia code lost its primary benefits, which were its flexibility and how closely the implementation mirrored the specification. In the LUT implementation, the Julia code uses almost the same syntax and verbiage as the C code, albeit with a slower execution speed. This does not necessarily disqualify Julia from use in FEC coding.

9.3.3 Hadamard Codes

Without a reference implementation, the Hadamard decoder was tested for throughput only for completeness. With k=3, the Julia implementation had a throughput of 1.24 Mbps, which appeared rather low. Since the decoder implementation is naïve, it was originally assumed that the decoder required most of the runtime, but subsequent experimentation showed that execution time was split nearly equally between encoding and decoding.

Extending the codeword size to k=8 caused the throughput to drop to 63.4 Kbps, which reveals the primary weakness of the Hadamard code, that the complexity is exponential versus k. While there are certainly more highly optimized versions of the Hadamard code than the one implemented here, there is not was to overcome the unfavorable complexity of the code itself.

9.3.4 Golay Codes

Liquid DSP has the facility to perform an extended Golay-(24,12) code. A simple benchmarking program was written in C++, requiring 36 non-blank, non-comment LOC. It was compiled as previously described using either "-O0" or "-O3" compiler flags. This benchmarking program was compared to a Julia implementation in Jupyter, which required only twelve lines of code to implement. As with the Hamming code, generation of the message words was moved outside the main task loop. The data are as follows:

Program	**C++ with -O0**	**C++ with -O3**	**Julia**
Throughput (Mbps)	102	143	10.3
Standard Dev	1.52	14.9	0.191
Coefficient of Variation	1.48%	10.4%	1.86%
Confidence Interval (Mbps)	1.89	18.5	0.238

Table 4: Throughputs for C/C++ and Julia programs implementing the Golay-(24,12) code.

The same notes apply to implementing the benchmarks for the Golay code as the Hamming codes, above.

As can be seen from **Table 4**, the C/C++ program was at least an order of magnitude faster than the Julia program, rising to a 14-fold throughput increase for the most optimized version.

Due to the complexity of the Golay decoder, it was suspected that this may be causing a slowdown in the Julia code. Closer inspection of the specification showed several areas where implementing the decoder faithfully would cause inefficiencies in the resulting algorithm that could be removed without impacting the correctness of the output. Measuring encoding only, the Julia code nearly doubled its throughput relative to the C/C++ implementation, with the most-optimized C/C++ encoder only eight times as fast (data not shown).

After the initial implementation, it was noted that the first 12 bits of the Golay codeword are simply the message bits repeated. Considering this, an optimized version of the Golay encoder that computed the dot product of only the latter half of the generator matrix and appended the result to the input message word was created. Since the number of multiplications was halved, this implementation was expected to operate at about twice the throughput of the original implementation. Results showed its throughput to be 33% higher than the original implementation, or about 1/6 the rate of the most optimized C/C++ code (data not shown). This value is slightly lower than expected from the results of [30] [85] [86]. Using the "`@fastmath`" and "`@inbounds`" did not have a statistically significant effect on the throughput of the optimized decoder via Welch's T-test.

9.3.5 Reed-Solomon Codes

Since Liquid DSP does not implement its own Reed-Solomon codes, instead calling out to a library that uses SIMD instructions for maximum speed, and a similar functionality was implemented in Julia. Thus, the results of the Reed-Solomon tests are not "can a Julia implementation match a C implementation in execution speed" but instead "what is the overhead associated with calling a C library function in Julia." According to the Julia documentation [30], there should be no overhead.

The results of making library calls for Reed-Solomon codes from C and Julia are shown in **Table 5**. All three implementations have similar throughput rates. There is no significant difference between the mean throughput of any implementation was not significant by Welch's T-test.

Program	C++ with -O0	C++ with -O3	Julia
Throughput (Mbps)	46.5	46.8	46.7
Standard Dev	0.297	1.18	0.905
Coefficient of Variation	0.639%	2.51%	1.94%
Confidence Interval (Mbps)	0.369	1.46	1.12

Table 5: Throughputs for C/C++ and Julia calling the libfec library for Reed-Solomon codes.

In this test case, the claims of the Julia developers are supported: there is no significant overhead associated with calling shared library C/assembly code from Julia versus calling it from C directly. This is perhaps the most powerful aspect of the Julia programming language, that it can build directly from the thousands of person-hours of work done in C/C++ and similar languages to achieve the effect of an active development community and long history *without either of those claims necessarily being true*.

9.3.6 Convolutional Codes

Liquid DSP implements a wrapper around libfec for convolutional decoding but provides a C implementation of the encoder. While no Julia implementation was developed for the Viterbi decoder, one was created for convolutional encoding. For a half-rate code of constraint length 7, the throughput values in **Table 6** were obtained.

Program	C++ with -O0	C++ with -O3	Julia
Throughput (Mbps)	152.8	162.9	3.35
Standard Dev	3.02	4.16	0.03
Coefficient of Variation	1.97%	2.55%	0.92%
Confidence Interval (Mbps)	3.75	5.16	0.04

Table 6: Throughputs for C/C++ and Julia implementing a rate-1/2 convolutional code with constraint length 7.

In both cases, the C/C++ code was over 40 times the throughput of the Julia code. The internals of the algorithms used in both implementations were examined and were determined to be largely similar save for the method in which the parity of each convolution was calculated. In the C implementation, a LUT was used, while in the Julia implementation the parity was

calculated on-the-fly using Kernighan's method. This discrepancy was judged to be the primary cause of the difference in performance between the two implementations, and a trivial C library implementing the parity calculation as a LUT was created to be called by the Julia code.

Adding a parity LUT increased the throughput of the Julia code to 47.75 Mbps, a fifteen-fold increase. Using the more efficient parity calculation, the Julia code could sustain approximately 1/6 the throughput of the C/C++ code, which is lower than what was measured for the Hamming encoder but on par with what was measured when optimizing the Golay coder. As mentioned previously, the Julia code was expected to sustain 1/3 or greater of the throughput of C/C++ code, depending on the use case. Even using similar algorithms, however, this was not possible for the convolutional encoder.

Puncturing the convolutional code reduced the throughput of the C/C++ encoder to 108.91 Mbps regardless of optimization level, a drop of approximately 1/3. The naïve Julia implementation suffered a throughput loss of 46.7%, while the optimized implementation lost 74.6% of its throughput, indicating a significant inefficiency on the Julia puncturing code. One possible source is in how the return values of the puncturing are presented: in Liquid DSP output parameters are used via pointers, while in the Julia code the punctured arrays are returned from the puncturing function directly. It is not clear from the Julia documentation whether the Julia compiler supports copy elision and/or move-enabled types.

9.4 Further Exploration

Enabling high levels of optimization in GCC allows the compiler to make many changes to the code, including vectorizing, inlining, and loop unrolling (among many others, please see http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html). Julia has several macros that allow optimization of the code, including "`@fastmath`" which turns off checks for NaN and overflow/underflow during mathematical operations, and "`@inbounds`" which disables boundary checking in the code (again, among many options). It was though that the Julia compiler may not

employ high levels of optimization by default, so the optimized Golay encoder and convolutional encoder were retested with both optimizations enabled.

In the case of neither algorithm was the difference in throughput statistically significant by Welch's T-test. These optimizations do not increase the efficiency of the Julia code by a measurable degree for the algorithms tested, with optimized C/C++ code outperforming the Julia code in all tested instances.

10 CONCLUSIONS

As Fred Brooks of IBM once famously wrote in his *No Silver Bullet* essay in the aftermath of the failed OS/360 project, “there is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability, in simplicity." For a young and unfinished language (the most recent release at the time of this writing was v0.6.2), Julia is an excellent language that offers the ability to rapidly turn mathematical specifications into executable code. But it is not a silver bullet for digital signal processing, at least not within the realm of FEC.

Julia implementations of four FEC algorithms proved to require many fewer SLOC than similar functionality written in C or C++ while remaining closer to the original specification. Both are remarkable boons for the language, as they allow functionality to be added in a way that does not compromise on readability. Whether the language is as productive as Python remains to be seen; for this project the full support for many mathematical operations provided by Julia was helpful over what is available in Python without additional libraries, though overall Julia is much more restrictive of a language (it will not allow type promotion *even when safe*).

In all test cases, however, Julia performed significantly worse than C/C++ in terms of execution speed and data throughput. Part of this shortcoming was in the implementations, as the Julia implementations sought to faithfully implement the mathematical specification of each FEC algorithm, while the C implementation was designed for use on a resource-constrained system and was thus more heavily optimized. In the cases of repetition coding and a re-implementation of the Hamming code to match the C implementation, the Julia implementation was on the same order of magnitude of throughput as the C/C++ implementation.

Judging by the Julia micro-benchmarks, the Julia code was expected to be faster [96]. The relative performance of Julia versus C/C++ varies wildly by application, however [85] [86]. Complaints

have been made against the Julia benchmark results [87], and the use of specific benchmarks to predict the performance of a platform in other domains has itself been called into question [88]. It is likely that in a mature implementation, the Julia code could be brought within a factor of two of the C/C++ code.

Perhaps the most salient takeaway from this project is that the claims made in [30] about the ability of Julia code to call C library functions without any significant overhead has been supported by the results of this work. The Reed-Solomon encoder and decoder called the same library from both the C and Julia implementations and showed no significant difference in throughput. The convolutional encoder could not match the overall speed of the C implementation with similar algorithms, but calling C code at critical junctures did increase the throughput by fifteen-fold. Many DSP applications and libraries, FEC or otherwise, have been written in C and C++ over the years, and Julia can leverage all of them. Historically, Python and SWIG have been used to create wrappers around the lower-level code for ease of use, but there is no reason that Julia, which is likely faster, cannot perform the same functions.

The conclusion of this project is this: Julia has a distinct applicability to the field of FEC in terms of implementation and maintainability but does not necessarily offer any advantages over C/C++ in terms of execution speed. It is not a silver bullet but is a useful tool. Often it is forgotten that the purpose of programming is to solve problems and implement solutions, not use programming languages or any particular tool to do so. There are situations in FEC when using Julia over C/C++ is the correct decision, but there are situations where the reverse holds as well.

11 REFERENCES

[1] J. Nielsen, "Nielsen's Law of Internet Bandwidth," 5 April 1998. [Online]. Available: nngroup.com.

[2] R. Hartley, "Transmission of Information," *International Congress of Telegraphy and Telephony,* 1927.

[3] H. Nyquist, "Certain Topics in Telegraph Transmission Theory," *Transactions of the A.I.E.E.,* p. 617–644, 1928.

[4] C. E. Shannon, "Communication in the Presence of Noise," *Proceedings of the IRE,* pp. 10-21, 1949.

[5] R. Jian, "Channel Models: a Tutorial," 21 February 2007. [Online]. Available: http://www.cse.wustl.edu/~jain/cse574-08/ftp/channel_model_tutorial.pdf.

[6] D. O. Case and L. M. Given, Looking for Information, Bingley: Emerald Group Publishing, 2016.

[7] R. Hamming, "Error Detecting and Correcting Codes," *The Bell Systems Technical Journal,* 1950.

[8] NCSTSD, "Telecommunications: Glossary of Telecommunications Terms," General Services Administration Information Technology Service, 1996.

[9] A. Svoboda, "From Mechanical Linkages to Electronic Computers: Recollections from Czechoslovakia," *IEEE,* 1980.

[10] ACM, "Richard W. Hamming Additional Materials," 2012. [Online]. Available: https://amturing.acm.org/info/hamming_1000652.cfm.

[11] C. E. Shannon, "A Mathematical Theory of Communication," *The Bell System Technical Journal,* pp. 379–423, 623–656, 1948.

[12] M. J. Golay, "Notes on Digital Coding," *Proceedings of the IRE,* vol. 37, p. 675, 1949.

[13] Cisco, "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021," Cisco, 2017.

[14] CTIA, "Wireless Snapshot 2017," CTIA, 2017.

[15] IEEE 802.11 Working Group, "IEEE 802.11-1997: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.," IEEE, 1997.

[16] IEEE Standards Association, "IEEE 802.11n-2009—Amendment 5: Enhancements for Higher Throughput," IEEE, 2009.

[17] L. Null and J. Lobor, The Essentials of Computer Organization and Architecture, Jones and Bartlett Publishers: Sudbury, 2006.

[18] G. E. Moore, "Cramming more components onto integrated circuits.," *Electronics,* 1965.

[19] M. Kanellos, "CNET," 11 February 2003. [Online]. Available: https://www.cnet.com/news/moores-law-to-roll-on-for-another-decade/. [Accessed 15 March 2018].

[20] H. Sutter, "The Free Lunch is Over," Sutter's Mill, March 2004. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm. [Accessed 15 March 2018].

[21] H. Sutter, "Welcome to the Jungle," Sutter's Mill, 2012. [Online]. Available: https://herbsutter.com/welcome-to-the-jungle/. [Accessed 15 March 2018].

[22] G. Moore, "Gordon Moore: The Man Whose Name Means Progress, The visionary engineer reflects on 50 years of Moore's Law," *IEEE Spectrum: Special Report: 50 Years of Moore's Law (Interview),* 2015.

[23] M. Andreessen, "Why Software is Eating the World," *The Wall Street Journal,* 20 August 2011.

[24] B. L. C. D. F. &. S. C. Bloessl, "A GNU radio-based IEEE 802.15.4 testbed.," *GI/ITG KuVS Fachgespräch Drahtlose Sensornetze,* pp. 37-40, 2013.

[25] B. S. M. S. C. &. D. F. Bloessl, "Decoding IEEE 802.11 a/g/p OFDM in Software using GNU Radio," *Proceedings of the 19th annual international conference on Mobile computing & networking,* pp. 159-162, 2013.

[26] A. Feickert, "The Joint Tactical Radio System (JTRS) and the Army's Future Combat System (FCS): Issues for Congress," US DOD, 2005.

[27] S. McConnell, Code Complete II, Redmond: Microsoft Press, 2004.

[28] J. Ousterhout, "Scripting: higher level programming for the 21st Century," *Computer,* vol. 31, no. 3, pp. 23-30, 1998.

[29] D. M. Beazley, "WIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++," in *4th Annual Tcl/Tk Workshop*, Monterey, CA, 1996.

[30] J. Bezanson, A. Edelman, S. Karpinski and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review,* vol. 59, no. 1, pp. 65-98, 2017.

[31] A. Covert, How to Make Sense of Any Mess: Information Architecture for Everybody, Scotts Valley, CA: CreateSpace Independent Publishing Platform, 2014.

[32] G. Bateson, Steps to an ecology of mind, New York: Ballantine Books, 1972.

[33] E. Parker, "Information and society," in *Proceedings of a Conference on the Needs of Occupational, Ethnic, and other Groups in the United States*, Washington, D.C, 1974.

[34] M. J. Bates, "Fundamental Forms of Information," *Journal of the American Society for Information Science and Technology.,* vol. 57, no. 8, pp. 1033-1045, 2006.

[35] G. Miller, "Information Theory in Psychology," in *The Study of Information: Interdisciplinary Messages*, New York, NY, Wiley, 1983, pp. 493-496.

[36] National Communications System, "Federal Standard 1037C," National Communications System, 2000.

[37] H. Balakrishnan and G. Verghese, 6.02 Introduction to EECS II: Digital Communication Systems, Cambridge: Massachusetts Institute of Technology, 2012.

[38] N. A. Armstrong, "The Engineered Century," *National Academy of Engineers: The Bridge,* vol. 30, no. 1, 2008.

[39] ITU, "BS.450 : Transmission standards for FM sound broadcasting at VHF," International Telecommunications Union, Geneva, Switzerland, 2001.

[40] SparkFun, "Analog vs. Digital," SparkFun, 18 July 2013. [Online]. Available: https://learn.sparkfun.com/tutorials/analog-vs-digital. [Accessed 8 April 2018].

[41] B. Sklar, Digital Communications, Second Edition, Upper Saddle River, NJ: Prentice Hall, 2017.

[42] ITU, "Generic framing procedure," August 2016. [Online]. Available: https://www.itu.int/rec/T-REC-G.7041-201608-I/en. [Accessed 9 April 2018].

[43] A. B. 2. Downey, Think DSP, Sebastopol, CA: O'Reilly, 2016.

[44] M. Viotti, "Mars Exploration Rovers," NASA Jet Propulsion Laboratory, Unknown. [Online]. Available: https://mars.jpl.nasa.gov/mer/credits/. [Accessed 16 March 2018].

[45] J. W. Perry, A. Kent and M. M. Berry, "Machine literature searching X. Machine language; factors underlying its design and development," *American Documentation,* vol. 6, no. 4, p. 242, 1955.

[46] E. N. Gilbert, "Capacity of a burst-noise channel," *Bell System Technical Journal,* vol. 39, pp. 1253-1265, 1960.

[47] B. Vucetic and J. Yuan, Turbo codes: principles and applications, Berlin: Springer Science+Business Media, 2000.

[48] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman and V. Stemann, "Practical Loss-Resilient Codes," in *Proc. 29th annual Association for Computing Machinery (ACM) symposium on Theory of computation*, 1997.

[49] F. Alt, "A Bell Telephone Laboratories' Computing Machine," *Mathematical Tables and Other Aids to Computation,* vol. 3, pp. 1-13, 60-84, 1948.

[50] S. Sparks and R. Kreer, "Tape Relay System for Radio Telegraph Operation," *R.C.A. Review,* vol. 8, pp. 393-426, 1947.

[51] T. M. Thompson, From Error-Correcting Codes through Sphere Packings to Simple Groups, The Carus Mathematical Monographs #21, The Mathematical Association of America, 1983.

[52] V. Guruswami, "List decoding of binary codes–a brief survey of some recent results," in *International Conference on Coding and Cryptology*, Berlin, DE, 2009.

[53] H. Dell and D. van Melkebeek, "CS 880: Pseudorandomness and Derandomization, Lecture 3: Error Correcting Codes," 30 January 2013. [Online]. Available: http://pages.cs.wisc.edu/~dieter/Courses/2013s-CS880/Scribes/PDF/lecture03.pdf. [Accessed 12 April 2018].

[54] P. Symonds, "MATH32031: Coding Theory, Part 4: Sphere Packing," University of Manchester, 5 September 2007. [Online]. Available: http://www.maths.manchester.ac.uk/~pas/code/part4.pdf. [Accessed 12 April 2018].

[55] A. Rudra, "Lecture 4: Hamming code and Hamming bound," University at Buffalo, August 2007. [Online]. Available: https://www.cse.buffalo.edu/faculty/atri/courses/coding-theory/lectures/lect4.pdf. [Accessed 12 April 2018].

[56] Defense Information Systems Agency, "MIL-STD-188," Defense Information Systems Agency, Washington, D.C., 2017.

[57] M. Malek, "Coding Theory: Golay Codes," Unknown. [Online]. Available: http://www.mcs.csueastbay.edu/~malek/Class/Golay.pdf. [Accessed 5 April 2018].

[58] E. R. Berlekamp, Key Papers in the Development of Coding Theory, Boca Raton, FL: CRC Press, 2001.

[59] 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.

[60] EBU, "Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for 11/12 GHz satellite services," European Broadcasting Union, Sophia Antipolis, FR, 1997.

[61] R. Singleton, "Maximum distance q-nary codes," *IEEE Trans. Inf. Theory,* vol. 10, no. 2, pp. 116-118, 1964.

[62] L. Welch, *The original view of reed-solomon codes,* 1997.

[63] E. R. Berlekamp, "Binary BCH decoding," in *International Symposium on Information Theory*, San Remo, Italy, 1967.

[64] J. L. Massey, "Shift-register synthesis and BCH decoding," *IEEE Trans. Information Theory,* vol. 15, no. 1, pp. 122-127, 1969.

[65] S. H. C. J. S. R. A. E. P. F. E. S. J. C. O. A. J. Caldwell, "Processing and Transmission of Information," Research Laboratory of Electronics (RLE) at MIT, Cambridge, MA, 1955.

[66] V. Lakkundi and M. Kasal, "FEC for Satellite Data Communications: Towards Robust Design," *WSEAS Transactions on Computers,* vol. 3, no. 6, pp. 2058-2061, 2004.

[67] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory,* vol. 13, no. 2, pp. 260-269, 1967.

[68] X. B. S. E. N. F. C. F. G. &. V. O. Anguera, "Speaker Diarization: A Review of Recent Research," *IEEE Transactions on Audio, Speech, and Language Processing,* vol. 20, no. 2, pp. 256-370, 2012.

[69] V. Pless, "Introduction to the Theory of Error-Correcting Codes," in *Wiley Series in Discrete Mathematics and Optimization*, Hoboken, NJ, John Wiley & Sons, 2011.

[70] Y. &. C. K. M. Feria, "Seamless data-rate change using punctured convolutional codes for time-varying signal-to-noise ratio.," in *IEEE International Conference on Communications*, Seattle, WA, 1995.

[71] C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," in *Proceedings of IEEE International Communications Conference*, Geneva, Switzerland, 1993.

[72] R. G. Gallager, "Low-Density Parity-Check Codes," in *Doctoral Thesis*, Cambridge, MA, 1963.

[73] D. J. MacKay, "Near Shannon Limit Performance of Low Density Parity Check Codes," *Electronics Letters,* 1996.

[74] B. Boehm, Software Engineering Economics, Englewood Cliffs, N.J.: Prentice Hall, 1981.

[75] J. Ousterhout, "Additional Information for Scripting White Paper," 1988. [Online]. Available: http://www.scriptics.com/people/john. ousterhout/scriptextra.html. [Accessed 20 March 2018].

[76] C. Jones, "Programming Languages Table, Release 8.2," The Lumen, March 1996. [Online]. Available: http://www.spr.com/library/0langtbl.htm. [Accessed 20 March 2018].

[77] R. Klepper and D. Bock, "Third and Fourth Generation Programming Language Productivity Differences," *Communications of the ACM,* vol. 38, no. 9, pp. 69-79, 1995.

[78] S. McConnell, Rapid Development: Taming Wild Software Schedules, Redmond, WA: Microsoft Press, 1996.

[79] J. Visser, "SIG/TÜViT Evaluation Criteria Trusted Product Maintainability," SIG/TÜViT , 2016. [Online]. Available: https://www.sig.eu/files/en/018_SIG-TUViT_Evaluation_Criteria_Trusted_Product_Maintainability.pdf. [Accessed 23 March 2018].

[80] J. Visser, Building Maintainable Software, Sebastopol, CA: O'Reilley, 2016.

[81] A. Stefnik and S. Seibert, "An Empirical Investigation into Programming Language Syntax," *ACM Transactions on Computing Education,* vol. 13, no. 4, 2015.

[82] P. Denny and A. C. D. Luxton-Reilly, "Enhancing Syntax Error Messages Appears Ineffectual," in *Proceedings of the 2014 conference on Innovation & technology in computer science education*, Uppsala, Sweden, 2014.

[83] D. Crawford, "Why mobile web apps are slow," Sealed Abstract, 20 June 2013. [Online]. Available: http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/. [Accessed 20 March 2018].

[84] I. Gouy, *The Computer Language Benchmarks Game.*

[85] J. Kouatchou, "Basic Comparison of Python, Julia, Matlab, IDL and Java (2018 Edition)," NASA, 2018. [Online]. Available: https://modelingguru.nasa.gov/docs/DOC-2676. [Accessed 22 March 2018].

[86] J. Kouatchou, "Basic Comparison of Python, Julia, R, Matlab and IDL," NASA, 2017. [Online]. Available: https://modelingguru.nasa.gov/docs/DOC-2625. [Accessed 22 March 2018].

[87] J. F. Puget, "How To Make Python Run As Fast As Julia," IBM, 1 December 2015. [Online]. Available: https://www.ibm.com/developerworks/community/blogs/jfp/entry/Python_Meets_Julia_Micro_Performance?lang=en. [Accessed 22 March 2018].

[88] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach (Fifth Edition), Waltham, MA: Elsevier, 2012.

[89] P. Ratanaworabhan, B. Livshits, D. Simmons and B. Zorn, "JSMeter: Characterizing Real-World Behavior of JavaScript Programs," Microsoft, Redmond, WA, 2009.

[90] T. DeMarco and T. Lister, Peopleware: Productive Projects and Teams (3rd Edition), Boston, MA: Addison-Wesley Professional, 2013.

[91] J. Atwood, "New Programming Jargon," Coding Horror, 20 June 2012. [Online]. Available: https://blog.codinghorror.com/new-programming-jargon/. [Accessed 20 March 2018].

[92] E. Lippert, "What is “duck typing”?," Fabulous adventures in coding, 2 January 2014. [Online]. Available: https://ericlippert.com/2014/01/02/what-is-duck-typing/comment-page-1/. [Accessed 6 April 2018].

[93] J. Spolsky, "The Law of Leaky Abstractions," Joel on Software, 11 November 2012. [Online]. Available: https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/. [Accessed 22 March 2018].

[94] S. Cass, "The 2017 Top Programming Languages," IEEE Spectrum, 2017. [Online]. Available: https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages. [Accessed 22 March 2018].

[95] B. A. Cipra, "The Best of the 20th Century: Editors Name Top 10 Algorithms," *SIAM News,* vol. 33, no. 4, 2000.

[96] Julia Group, "Julia Micro-Benchmarks," Julia Development Team, Unknown. [Online]. Available: https://julialang.org/benchmarks/. [Accessed 7 March 2018].

[97] Computer Literacy Bookshops, Inc, "DONALD KNUTH-Computer Literacy Bookshops Interview," 1993. [Online]. Available: http://tex.loria.fr/litte/knuth-interview. [Accessed 22 March 2018].

[98] J. Kalb and A. Gasper, C++ Today: The Beast is Back, Sebastopol, CA: O'Reilley, 2015.

[99] H. Sutter, "Elements of Modern C++ Style," Sutter's Mill, 2011. [Online]. Available: https://herbsutter.com/elements-of-modern-c-style/. [Accessed 6 April 2018].

[100] S. O'Grady, "The RedMonk Programming Language Rankings: January 2018," RedMonk, 7 March 2018. [Online]. Available: http://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/. [Accessed 22 March 2018].

[101] TIOBE Company, "TIOBE Index for March 2018," TIOBE Company, March 2018. [Online]. Available: https://www.tiobe.com/tiobe-index. [Accessed 22 March 2018].

[102] C. Hoare, "Hints on Programming Language Design," Stanford Artificial Intelligence Laboratory, Memo AIM 224, Computer Science Department Report No. CS-403, Stanford, 1973.

[103] R. Chen, "Code is read much more often than it is written, so plan accordingly," Microsoft, 6 April 2007. [Online]. Available: https://blogs.msdn.microsoft.com/oldnewthing/20070406-00/?p=27343. [Accessed 27 March 2018].

[104] D. Luu, "A review of the Julia language," 2014. [Online]. Available: http://danluu.com/julialang/. [Accessed 27 March 2018].

[105] R. Chen, "Try to avoid having BOOL function parameters," Microsoft, 28 August 2006. [Online]. Available: https://blogs.msdn.microsoft.com/oldnewthing/20060828-18/?p=29953/. [Accessed 27 March 2018].

[106] R. Chen, "Cleaner, more elegant, and harder to recognize," Microsoft, 14 January 2005. [Online]. Available: https://blogs.msdn.microsoft.com/oldnewthing/20050114-00/?p=36693. [Accessed 27 March 2018].

[107] N. Yorav-Raphael, "Julia user group posting," 23 February 2012. [Online]. Available: https://groups.google.com/forum/?hl=en#!topic/julia-dev/tNN72FnYbYQ. [Accessed 27 March 2018].

[108] J. Domke, "Julia, Matlab, and C," 17 September 2012. [Online]. Available: https://justindomke.wordpress.com/2012/09/17/julia-matlab-and-c/#results. [Accessed 22 March 2018].

[109] R. W. Oliver, "Building a Website With C++," Sourcerer, 20 January 2018. [Online]. Available: https://blog.sourcerer.io/building-a-website-with-c-db942c801aee. [Accessed 22 March 2018].

[110] J. Atwood, "VB vs. C# -- FIGHT!," Coding Horror, 14 July 2004. [Online]. Available: https://blog.codinghorror.com/vb-vs-c-fight/. [Accessed 22 March 2018].

[111] J. Atwood, "VB vs. C#, round two," Coding Horror, 19 November 2004. [Online]. Available: https://blog.codinghorror.com/vbnet-vs-c-round-two/. [Accessed 22 March 2018].

[112] J. Atwood, "Stuck in a VB.NET Ghetto," Coding Horror, 13 October 2004. [Online]. Available: https://blog.codinghorror.com/stuck-in-a-vbnet-ghetto/. [Accessed 22 March 2018].

[113] J. Atwood, "The Slow Brain Death of VB.NET," Coding Horror, 10 March 2005. [Online]. Available: https://blog.codinghorror.com/the-slow-brain-death-of-vb-net/. [Accessed 22 March 2018].

[114] Y. Minsky and S. Weeks, "Caml trading – experiences with functional programming on wall street," *Journal of Functional Programming,* vol. 18, no. 4, pp. 553-564, 2008.

[115] A. Claster, "Julia Ranks Among Top 10 Programming Languages Developed on GitHub," Julia Development Team, 25 March 2017. [Online]. Available: https://juliacomputing.com/press/2017/05/25/github-top-ten.html. [Accessed 22 March 2018].

[116] S. D. D'Cunha, "How A New Programming Language Created By Four Scientists Now Used By The World's Biggest Companies," Forbes, 20 September 2017. [Online]. Available: https://www.forbes.com/sites/suparnadutt/2017/09/20/this-startup-created-a-new-programming-language-now-used-by-the-worlds-biggest-companies/#1faf6a467de2. [Accessed 27 March 2018].

12 APPENDIX 1: EXAMPLE JULIA CODE FOR HAMMING CODE

```
module Hamming

################################################################################
# INCLUDES
################################################################################
include("types.jl")

################################################################################
# EXPORTS
################################################################################
export Hamming_Params
export construct_A, construct_G_from_A, construct_H_from_A,
construct_R_from_A, construct_hamming_parameters
export encode, decode, encode_stream, decode_stream

################################################################################
# TYPES
################################################################################
type Hamming_Params
    A::BitArray
    G::BitArray
    H::BitArray
    R::BitArray
end

################################################################################
# FUNCTIONS
################################################################################

"""
    construct_A( block_len::SizeType, message_len::SizeType )

Create a valid A matrix for a Hamming code of arbitrary size.  It is the
responsibility of
the caller to ensure that block_len and message_len are valid parameters for
a Hamming code.

Hamming codes are generally named \"Hamming-(block_len, message_len).\"
Common Hamming codes include:
* Hamming-(7,4)
* Hamming-(15,11)
* Hamming-(31,26)
* etc.
"""
function construct_A( block_len::SizeType, message_len::SizeType )
    @assert block_len > message_len

    # r is the number of parity bits to be added
    r = block_len - message_len
```

```
    # Each parity bit 'n' will be placed in column 2^(n-1)
    # These columns are exised in A, so we need to keep track of them
    parity_cols = [ 2^(x-1) for x in range(1,r) ]

    # Data columns include all columns in A except parity columns
    data_cols = find( [ !(x in parity_cols) for x in range(1,block_len) ] )

    # Each parity bit 'n' protects the codeword bits in columns where the bit
corresponding
    # to 'n' is set.  So parity bit 1 protects all odd columns, etc.
    parity_masks = [ (1 << x) for x in range(0,length(parity_cols)) ]

    # First make the whole table, then excise parity columns later
    A_init = zeros( Bit, r, block_len )

    for row = 1:size(A_init,1)
        for column = 1:size(A_init,2)
            # Is this column protected by the parity bit in question?
            A_init[row,column] = ( column & parity_masks[row] ) > 0
        end
    end

    # And now excise the parity columns
    A = zeros( Bit, r, message_len )
    for d = 1:length(data_cols)
        A[:,d] = A_init[:,data_cols[d]]
    end

    return A
end

"""
    construct_G_from_A( A::BitArray )

Create a genrator matrix for a Hamming code of arbitrary size from a
previously-construct A
matrix.  It is easiest to pass the return value of construct_A( block_len,
message_len ) as
the parameter to this function.
"""
function construct_G_from_A( A::BitArray )
    # All parameters used to construct A can be retreived from its structure
    r = size(A,1)
    block_len = 2^r - 1
    message_len = 2^r - r - 1

    parity = A
    data = eye( Bit, message_len )

    # Since G is being constructed in its transposed form, we think of parity
bits as
    # protecting rows, not columns
    parity_rows = [ 2^(x-1) for x in range(1,r) ]
```

```
    # The data rows are all the non-parity rows
    data_rows = find( [ !(x in parity_rows) for x in range(1,block_len) ] )

    G = zeros( Bit, block_len, message_len )

    for p = 1:size(parity,1)
        G[parity_rows[p],:] = parity[p,:]
    end

    for d = 1:size(data,1)
        G[data_rows[d],:] = data[d,:]
    end

    return G
end

"""
    construct_H_from_A( A::BitArray )

Create a parity check matrix for a Hamming code of arbitrary size from a
previously-constructed
A matrix.  It is easiest to pass the return value of construct_A( block_len,
message_len ) as
the parameter to this function.
"""
function construct_H_from_A( A::BitArray )
    # All parameters used to construct A can be retreived from its structure
    r = size(A,1)
    block_len = 2^r - 1
    message_len = 2^r - r - 1

    data = A
    parity = eye( Bit, r )

    # Each parity column 'n' will be placed in column 2^(n-1)
    parity_cols = [ 2^(x-1) for x in range(1,r) ]

    # Data columns are all the non-parity columns
    data_cols = find( [ !(x in parity_cols) for x in range(1,block_len) ] )

    H = zeros( Bit, r, block_len )

    for p = 1:size(parity,2)
        H[:,parity_cols[p]] = parity[:,p]
    end

    for d = 1:size(data,2)
        H[:,data_cols[d]] = data[:,d]
    end

    return H
end
```

```
"""
    construct_R_from_A( A::BitArray )

Create a decoding matrix for a Hamming code of arbitrary size from a
previously-constructed
A matrix.  It is easiest to pass the return value of construct_A( block_len,
message_len ) as
the parameter to this function.
"""
function construct_R_from_A( A::BitArray )
    # All parameters used to construct A can be retreived from its structure
    r = size(A,1)
    block_len = 2^r - 1
    message_len = 2^r - r - 1

    data = eye(Bit, message_len)

    # Each parity column 'n' will be placed in column 2^(n-1)
    parity_cols = [ 2^(x-1) for x in range(1,r) ]

    # Data columns are all the non-parity columns
    data_cols = find( [ !(x in parity_cols) for x in range(1,block_len) ] )

    R = zeros( Bit, message_len, block_len )
    for d = 1:message_len
        R[:,data_cols[d]] = data[:,d]
    end

    return R
end

"""
    construct_hamming_parameters( block_len::SizeType, message_len::SizeType
)

Construct the A, G, H, and R matrices necessary for Hamming encoding and
decoding.
"""
function construct_hamming_parameters( block_len::SizeType,
message_len::SizeType )
    A = construct_A( block_len, message_len )
    G = construct_G_from_A( A )
    H = construct_H_from_A( A )
    R = construct_R_from_A( A )

    return Hamming_Params( A, G, H, R )
end
```

```
"""
    encode( G::BitArray, p::BitVector )

# Arguments
* `G::BitArray`: A Hamming generator Matrix
* `p::BitVector`: A message word with length equal to the number of columns
of G

Encode a single message word into a single codeword.  It is the
responsibility of the caller
to ensure that (1) G is valid, and (2) that P has the correct dimensions.
"""
function encode( G::BitArray, p::BitVector )
    return mod.( G * p, Bit(2) )
end

"""
    decode( H::BitArray, R::BitArray, r::BitVector )

# Arguments
* `H::BitArray`: A Hamming parity check Matrix
* `R::BitArray`: A Hamming error correction Matrix
* `r::BitVector`: A codeword with length equal to the number of columns of H

Decode a Hamming codeword 'r' to the original message.  If an error is
detected using the
parity check matrix H, it is corrected using the matrix R.  The location of
the error is
also returned to the caller, with a location of '0' indicating no error.

Note that Hamming codes can correct only one error per codeword, so if the
error rate is
higher than that, the decoding will fail silently.
"""
function decode( H::BitArray, R::BitArray, r::BitVector )

    # Internal function to correct an error in a codeword 'r' using the
syndrome 'z'
    function correct_error!( r::BitVector, z::BitVector )

        # Internal function to coerce a syndrome vector Z to an array index
        function z_to_int( z::BitVector )
            nbits = size(z, 1)
            ret = 0
            for i = 1:nbits
                ret |= ( z[i] << (i-1) )
            end

            return ret
        end # z_to_int

        error_location = z_to_int( z )
```

```
        # An error location of zero means no error
        if error_location > 0
            # Flip the error bit
            r[error_location] = xor( r[error_location], 1 )
        end

        return error_location
    end # correct_error!

    syndrome = mod.( H * r, Bit(2) )
    error_loc = correct_error!( r, syndrome )

    return ( mod.( R * r, Bit(2) ), error_loc )
end

end # module Hamming
```

Note: Code was colorized using http://hilite.me/ code beautifier.

13 APPENDIX 2: EXAMPLE JULIA CODE FOR REED-SOLOMON CODE

```
module rs_ccall

##############################################################################
# NOTES
##############################################################################

# 1. All functions in this file require libfec:
https://github.com/quiet/libfec
# 2. The return value of init_RS_state is an opaque C-type, not a Julia type

##############################################################################
# INCLUDES
##############################################################################
include("types.jl")

##############################################################################
# EXPORTS
##############################################################################
export init_RS_state, free_RS_state, rs_encode, rs_decode!

##############################################################################
# FUNCTIONS
##############################################################################
"""
    init_RS_state( symsize::Int32, gfPoly::Int32, fcr::Int32, prim::Int32,
nroots::Int32, pad::Int32 )

Create the necessary state for Reed-Solomon encoding.  The result of this
function
must be passed to the encode and decode functions for them to work properly.

Note that the return value of this function is an unmanaged resource and must
be
manually destroyed via free_RS_state( rs_state ).
"""
function init_RS_state( symsize::Int32, gfPoly::Int32, fcr::Int32,
prim::Int32, nroots::Int32, pad::Int32 )
    return ccall((:init_rs_char, "libfec"), Ptr{Void},
(Int32,Int32,Int32,Int32,Int32,Int32), symsize, gfPoly, fcr, prim, nroots,
pad )
end
```

```
"""
    free_RS_state( rs_state )

Destroy the state used for a Reed-Solomon encoder and free the associated
memory.
Must be called before exiting a program that has called init_RS_state().
"""
function free_RS_state( rs_state )
    ccall((:free_rs_char, "libfec"), Void, (Ptr{Void},), rs_state)
end

"""
    rs_encode( unencoded::ByteVector, rs_state )

Encode a block of 223 bytes with a Reed-Solomon-(255,223) code.
"""
function rs_encode( unencoded::ByteVector, rs_state )
    @assert( length(unencoded) == 223 )
    parity = zeros( Byte, 32 )
    ccall( (:encode_rs_char, "libfec"), Ptr{Void}, (Ptr{Void},
Ptr{UInt8},Ptr{UInt8}), rs_state, unencoded, parity )
    return vcat( unencoded, parity )
end

"""
    rs_decode!( encoded::ByteVector, error_locs::Vector{Int32}, rs_state )

Decode a block of encoded Reed-Solomon-(255,223) data and correct any errors.
The argument error_locs may be (any usualy is) the zero vector, and may be
modified
by the decoder to show the presence of errors.  The encoded message is
modified
iff errors are found, with the output being the corrected vector.
"""
function rs_decode!( encoded::ByteVector, error_locs::Vector{Int32}, rs_state
)
    @assert( length(encoded) == 255 )
    ccall( (:decode_rs_char, "libfec"), Ptr{Void}, (Ptr{Void}, Ptr{UInt8},
Ptr{Int32}, Int32), rs_state, encoded, error_locs, 0 )
    return encoded[1:223]
end

end # module rs_ccall
```

Note: Code was colorized using http://hilite.me/ code beautifier.