3-1-2018

# Techniques for Low-latency in Software-defined Radio-based Networks

Daniel D. Hart


# TECHNIQUES FOR LOW-LATENCY IN SOFTWARE-DEFINED RADIO-BASED NETWORKS

THESIS

Daniel Hart, Capt, USAF

AFIT-ENG-MS-18-M-032

**DEPARTMENT OF THE AIR FORCE**
**AIR UNIVERSITY**

***AIR FORCE INSTITUTE OF TECHNOLOGY***

**Wright-Patterson Air Force Base, Ohio**

AFIT-ENG-MS-18-M-032

# TECHNIQUES FOR LOW-LATENCY IN SOFTWARE-DEFINED RADIO-BASED NETWORKS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Science

Daniel Hart, B.S.A.E.

Capt, USAF

March 2018

AFIT-ENG-MS-18-M-032

# TECHNIQUES FOR LOW-LATENCY IN SOFTWARE-DEFINED RADIO-BASED NETWORKS

THESIS

Daniel Hart, B.S.A.E.
Capt, USAF

Committee Membership:

Dr. Kenneth Hopkinson
Chair

Maj Addison Betances
Member

Maj Daniel Casey
Member

AFIT-ENG-MS-18-M-032

# Abstract

In recent years, the role the United States Air Force (USAF) plays in defending the nation's security and sovereignty and the missions for which it is responsible have increased. In addition, decreased budgets and the proliferation of wireless devices present major challenges the force must confront moving forward. As a result, the USAF must apply existing systems and capabilities in innovative ways to ensure mission accomplishment subject to resource limitations. One such new application is the use of Unmanned Aerial Vehicle (UAV) swarms in missions previously performed by single, larger UAV platforms. Swarms of smaller UAV platforms offer increased flexibility and resilience as compared to their larger counterparts.

One problem with the increased utilization of UAV swarms, however, is the congestion of the electromagnetic spectrum through which they communicate. The advent of the Internet of Things (IoT) could present swarms with unintentional or intentional interference with which they must contend in future missions. In light of this, a solution which offers robust and reliable swarm communication in spite of the increased electromagnetic spectrum congestion is needed. One promising technology that could form the basis of a potential solution is the Software-Defined Radio (SDR). The core of an SDR's functionality is implemented in software, allowing the user to alter its functionality while it is operating. Further, SDRs enable fully cognitive radios which have an Artificial Intelligence (AI) back-end to allow them to autonomously react and adapt to their transmission environment with no user interaction.

The present research aims to develop and test a Genetic Algorithm (GA)-based Cognitive Engine (CE) to begin looking at real-time CEs that could be used in future UAV swarms. A real-time engine is one that executes while the network is running

and periodically updates operating parameters to tune the network's performance to its current environment. As UAV swarms are the long-term application for this research, communication latency is the optimization objective of primary importance in tuning network performance here.

In testing the engine, particular items of interest include the number of solutions it may evaluate in a given bound and the engine's reliability in returning solutions which yield acceptable network performance. Initial experiments indicate that, subject to design and implementation decisions, the engine can consider significant portions of the search space within a relatively small bound. These tests also indicate the engine is efficient at finding highly fit solutions in terms of network latency, throughput and power consumption fitness. However, additional testing is needed to evaluate how closely highly fit solutions produced by the engine correlate to reduced latency within the network.

There are several potential directions in which this research could be taken in the future. For example, the engine could be updated to take into account additional network operating parameters. Another possibility is to assign different weights to the fitness components or test another fitness function to evaluate how these changes affect the engine's performance. A third option is test the engine with noise floors beyond those used here to determine how increased noise affects the engine's performance.

# Table of Contents

Page

Page

# List of Figures

# List of Tables

# List of Abbreviations

| Abbreviation | | Page |
|---|---|---|
| CEs | Cognitive Engines | 10 |
| FPGAs | Field-Programmable Gate Arrays | 11 |
| GPUs | Graphics Processing Units | 11 |
| MIMO | Multiple-Input Multiple-Output | 11 |
| FEC | Forward Error Correction | 11 |
| GPP | General-Purpose Processor | 12 |
| GA | Genetic Algorithms | 14 |
| AP | Access Point | 14 |
| CBR | Case-Based Reasoning | 15 |
| DTs | Decision Trees | 15 |
| CBR-QGA | Case-Based Reasoning-Quantum Genetic Algorithm | 16 |
| QGA | Quantum Genetic Algorithm | 16 |
| CRM | Cognitive Resource Manager | 17 |
| CellBE | Cell Broadband Engine | 17 |
| DRM | Distributed Resource Map | 18 |
| P | Polynomial-time | 19 |
| NP | Nondeterministic Polynomial | 19 |
| SNR | signal-to-noise ratio | 29 |
| PSK | Phase-Shift Keying | 30 |
| QAM | Quadrature-Amplitude Modulation | 30 |
| NS-2 | Network Simulator 2 | 33 |
| NS-3 | Network Simulator 3 | 33 |
| OMNeT++ | Object Modular Network Testbed in C++ | 33 |

| Abbreviation | | Page |
|---|---|---|
| CORNET | COgnitive Radio NETwork | 34 |
| SSH | Secure SHell | 34 |
| CRTS | Cognitive Radio Test System | 34 |
| MATLAB | MATrix LABratory | 34 |
| CE | Cognitive Engine | 35 |
| UAVs | Unmanned Aerial Vehicles | 35 |
| IP | Internet Protocol | 37 |
| MAC | Medium Access Control | 37 |
| TCP | Transmission Control Protocol | 37 |
| UDP | User Datagram Protocol | 37 |
| SCTP | Stream Control Transmission Protocol | 37 |
| NICs | network interface cards | 38 |
| QoS | quality-of-service | 38 |
| CSMA-CA | Carrier-Sense Multiple Access with Collision Avoidance | 38 |
| AI | Artificial Intelligence | 39 |
| BER | bit error rate | 42 |
| QAM | Quadrature Amplitude Modulation | 43 |
| PSK | Phase Shift Keying | 43 |
| FEC | Forward Error Correction | 52 |
| CRTS | Cognitive Radio Test System | 56 |
| FEC | Forward Error Correction | 56 |
| CE | Cognitive Engine | 56 |
| QAM | Quadrature Amplitude Modulation | 57 |

| Abbreviation | | Page |
|---|---|---|
| ASK | Ampitude Shift Keying | 57 |
| PSK | Phase Shift Keying | 57 |
| NTP | Network Time Protocol | 59 |
| BERs | Bit Error Rates | 67 |
| PNG | pseudo-random number generator | 69 |
| AWGN | Additive White Gaussian Noise | 71 |
| SNR | Signal-to-Noise Ratio | 73 |
| UAV | Unmanned Aerial Vehicle | 74 |
| ASK | Amplitude Shift Keying | 81 |
| UAV | Unmanned Aerial Vehicle | 92 |
| FEC | Forward Error Correction | 92 |
| SNR | Signal-to-Noise Ratio | 95 |
| BER | Bit Error Rate | 95 |
| BCH | Bose-Chaudhuri-Hocquenghem | 96 |
| PSK | Phase Shifft Keying | 110 |
| QAM | Quadrature Amplitude Modulatio | 110 |
| CE | Cognitive Engine | 123 |
| FEC | Forward Error Correction | 123 |
| BERs | Bit Error Rates | 124 |
| QoS | Quality of Service | 125 |
| SNR | Signal-to-Noise Ratio | 125 |
| UAV | Unmanned Aerial Vehicle | 127 |

TECHNIQUES FOR LOW-LATENCY IN SOFTWARE-DEFINED RADIO-BASED NETWORKS

# I. Introduction

## 1.1 Background

Due to ongoing fiscal realities within the Department of Defense (DoD), the United States Air Force (USAF) must continually innovate ways to do more with less. Force readiness and modernization are key issues and the USAF must ensure every dollar is spent with maximum effectiveness [1]. In 2014, the USAF recognized the force's ability to quickly adapt to an operational environment that itself undergoes rapid change as vital to maintaining the nation's strategic advantage in the future. The USAF has further identified the continued development of unmanned and autonomous systems as crucial to the increased agility demanded by the future fight [2]. Two ways in which the USAF is seeking to optimize its spending while maximizing agility is employing its existing systems in new ways and adapting commercial solutions to its needs. By using existing systems in new roles, operators have greater flexibility in ensuring mission accomplishment. Further, effectively adapting existing commercial solutions to its needs allows the USAF to save on development costs while more rapidly fielding required capabilities, allowing the force to quickly respond to emerging threats.

Representative examples of these trends include the use of swarms of smaller Unmanned Aerial Vehicles (UAVs) in missions previously performed by a single, larger platform and the proposal to use Software-Defined Radios (SDRs) to provide intra-

swarm communications in future missions [3, 4, 5]. The use of UAV swarms offers potential cost savings to the USAF. The vehicles that typically comprise a swarm are smaller and cheaper to build, maintain, operate and replace than their larger counterparts. By extension, this reduces operational risk; losing even a handful of smaller UAV platforms is not as costly as the destruction of a larger platform. Further, a UAV swarm can execute new mission types or existing mission types in new ways not possible with a single, larger platform. For example, a UAV swarm could fly in formation to the boundary of an adversary's air defense system, scatter in such a manner as to avoid detection by or overwhelm air defense radar systems and then reassemble itself once the air defense sector has been penetrated. As another example, each platform in a swarm could be outfitted with a sensor suite to ensure continual Intelligence, Surveillance and Reconnaissance (ISR) coverage in a contested environment. If one or more platforms in the swarm is destroyed, the other vehicles ensure continual coverage, a robust mission capability not offered by a single platform performing ISR missions [6].

A UAV swarm that relies on SDRs for intra-swarm communications could provide the basis for a "cognitive" UAV swarm. Such a swarm could adapt its operating parameters to its environment to optimize intra-swarm communication in much the same way nodes in a cognitive network alter their parameters to optimize performance. Before an SDR-based or fully cognitive UAV and UAV swarm may be realized and fielded, however, there exist significant technical challenges that must be overcome. These challenges include the development of a new, or adaptation of an existing, SDR system to a UAV platform. While there are numerous SDR systems available off-the-shelf, the UAV platforms proposed for use in a swarm are typically small platforms. It may be difficult or impossible to retrofit existing SDR systems on board these UAV platforms due to size or weight constraints, especially when the necessary power

supplies are taken into account. Perhaps more importantly, a considerable amount of research remains to be done to determine the most efficient and effective algorithms for use in adapting a swarm to its environment. Numerous performance parameters, the dependent optimization objectives and the large number of paradigms available for use in developing such algorithms yield a vast number of possible solutions. Only a small subset of this search space has thus far been explored. It remains to be seen which algorithms will ultimately provide the optimal performance within a cognitive swarm.

## 1.2 Genetic-Algorithm Based Cognitive Engine Development and Testing

In the present research, a cognitive network simulated using the OMNet++ Network Simulator is used as a substitute for a real-world cognitive UAV swarm. While throughput is often the optimization objective of interest, latency is the performance metric under consideration here. Within a cognitive UAV swarm, low-latency is likely to be a critical requirement in intra-swarm communication. In order for the platforms to accomplish the mission, messages sent between them will likely need to be delivered with an upper bound on their delay. For example, if one of the platforms in the swarm is to strike a target, rules of engagement may require the firing platform confirm its location with another swarm member before employing ordinance. If no guarantee can be given on the time frame in which the first platform may expect a response, the opportunity to strike the designated target may be missed. Here, a Genetic Algorithm (GA)-based Cognitive Engine (CE) which alters the modulation scheme and order, transmitter power and bandwidth in an attempt to counter intentional and unintentional interference is developed and tested. The evaluation of the CE will focus on determining whether or not latency may serve as an acceptable performance parameter in tuning the network parameters to optimize performance.

In particular,

- How may network latency information be incorporated into the CE in such a manner that all necessary information is available to the engine when it executes its decision algorithm?
- What is the extent to which tuning modulation scheme and order, transmitter power and bandwidth may affect network latency and is this effect large enough that these parameters may serve as a viable basis for future UAV swarm CEs?
- Is the time required by a genetic algorithm-based CE so large as to render it ineffective in tuning network performance and thus unacceptable as a basis for a UAV swarm CE?
- How does the fitness of solutions returned by the engine vary as a function of the engine's runtime? Does allowing the engine to run longer before returning a solution reliably yield more fit responses?

## 1.3 Thesis Organization

The next chapter explores several of the terms and concepts necessary for understanding the discussion of the network and CE development as well as the experiments performed to test the engine. Chapter 3 briefly discusses the architecture of the simulated network in which the CE is tested and details the development of the engine itself. Then, Chapter 4 outlines the experiments conducted using the network to test the CE's performance. Finally, Chapter 5 presents the analysis of the data obtained from the experiments and Chapter 6 concludes with final recommendations drawn from the experimental results.

# II. Background and Related Research

## 2.1 Introduction

With the rise of the Internet of Things (IoT) and the proliferation of wireless networking technologies, the electromagnetic frequency bands used to carry communications between devices have become overcrowded [7]. This overcrowding is due in large part to the inefficient use of available spectrum resources. Historically, frequencies have been leased to a Primary User (PU) and no transmitters have been permitted to operate within bands leased to another user. While such an approach ensures PUs may always communicate when necessary, unless they transmit for a considerable percentage of the time, their spectrum band goes largely unused, leading to inefficient use of available spectrum resources [8]. As a result, as the available frequency bands on which to transmit dwindle, without solutions that better utilize spectrum resources, new PUs may not be able to obtain a spectrum allocation. Fortunately, cognitive radios have been proposed as one solution that may help mitigate the inefficient use of spectrum resources. In this new approach, a Secondary Users (SU) is a user to whom a frequency band has not been officially licensed but is allowed to opportunistically use it as long as they do not interfere with the PU. SUs use cognitive radio-based nodes to transmit and periodically listen for the PU to vacate the channel if the PU begins to transmit [9]. This acts to increase the percentage of time frequency bands are used and thus spectral efficiency.

The ability to exchange information between platforms in an Unmanned Aerial Vehicle (UAV) swarm is crucial to the continued and future success of UAV swarm missions. For example, operators may want more than one platform to confirm the swarm is in the correct location before employing munitions or engaging in surveillance and/or reconnaissance activities. Platforms may also need to be able to compare

sensor readings as part of the swarm's mission or to confirm swarm health or operational parameters (e.g., temperature). Often, this information has hard limits on the time in which it must be exchanged. In light of these needs, reliable and low-latency communication will be vital to the continued employment of UAV swarms to accomplish United States Air Force (USAF) missions. However, with electromagnetic spectrum quickly filling up, the increasing contention for limited available spectrum could threaten intra-swarm communication in the future. Flexible UAVs that can adapt their transmission protocols and parameters to their local operating environment could help ensure reliable communication channels.

While communications within future swarms could be based on existing wireless technologies, interference on the frequency or small number of frequencies used by those technologies could lead to critical communications failures. Conversely, communications based on Software-Define Radios (SDRs) or Cognitive Radios (CRs) could help mitigate this risk by allowing the swarm to dynamically adapt its operating parameters to effectively counter interference and safeguard communications. While this is an important capability for future missions, there are further benefits of using SDR- or CR-based communications. The ability to adapt operating parameters to environmental conditions or transmission needs allows the swarm to more efficiently use other resources. For example, if a low data rate, short-range link is needed, using transmission parameters that support such a link rather than a high-bandwidth, high-power link conserves power.

## 2.2 Cognitive Radio Networks

Although the term "cognitive radio networks" is often used to refer to networks in which SUs opportunistically use frequency bands licensed to other PUs, this conception of a cognitive radio network actually falls far short of cognitive networking

as originally proposed. The term "cognitive radio" was first coined by Joseph Mitola in his 2000 dissertation [10]. He envisioned future networks in which mobile nodes exchange information about network conditions and their own computational needs to better utilize available resources (e.g., spectrum bandwidth). Using an Artificial Intelligence (AI) component (the "cognitive" in "cognitive network"), these nodes ingest and interpret environmental information far beyond the local transmission environment to estimate and provide for their future resource needs. Additionally, the nodes in the network cooperate to evolve their operating parameters and protocols to improve network efficiency and user experience over time [11]. While the network administrators set parameter values and algorithms with which the nodes initially operate, the nodes themselves work together to alter these parameters and algorithms in an effort to find better ways to provide network services.

To help illustrate cognitive networking as originally envisioned, suppose a smartphone owner, whose device was designed for cognitive networking, uses their phone to purchase tickets for a movie that is showing in two hours at a theater across town. The smartphone would be able to predict that in approximately an hour and half, the user will transit from their current location to the theater. It may also be able to predict the routes the user may take with different probabilities. The device may then communicate this information to surrounding nodes and the supporting network infrastructure. In a fully cognitive network where all nodes are predicting the amount and location of their future resource needs, the network may use this information to optimize the distribution of resources. Interestingly, the environmental awareness necessary to enable these services is already appearing on smart devices. For example, a smartphone running Google's Calendar application will notify the user when they have an upcoming appointment and will give them a prediction of how it will take them to reach their destination.

As a simple example of how a cognitive network may evolve over time, consider the wireless network shown in Figures 1 and 2 where Nodes A and B are much closer to the base station than Nodes C and D. As a result, Nodes A and B do not need to transmit with as much power as Nodes C and D to be received at the base station. However, suppose the network administrators initially configure all four nodes to transmit at the same power. As a result, the base station may miss transmissions from Nodes C and D because one or both of Nodes A and B is simultaneously transmitting and the base station cannot hear Nodes C and D over Nodes A and B. If the radio nodes and base station were cognitive nodes, they would be able to communicate about their transmission parameters and their realized performance. Over time, they may discover that decreasing the transmission power of Nodes A and B increases the throughput by allowing Nodes C and D to still be heard when Node A or B is transmitting and thus allow the nodes to transmit simultaneously. (Admittedly, such a scenario would require the nodes to be on different frequencies and the base station to have more than one receiver channel.)

As a result, the network may make the decision to permanently alter the transmission power of Nodes A and B. This has the added benefit of making the network more energy efficient; with Nodes A and B using less power to transmit, the network does not require as much energy to run as before. This could be especially important if the power sources on which the nodes run have a finite amount of energy (e.g., batteries) before they must be replaced or recharged. In this hypothetical scenario, neither the increased throughput nor the energy savings would be possible without cognitive networking.

As stated previously, cognitive radios and networks as designed and implemented to this point have fallen considerably short of Mitola's original vision. Most of the networks realized to date have focused on conditions in the local operating environment

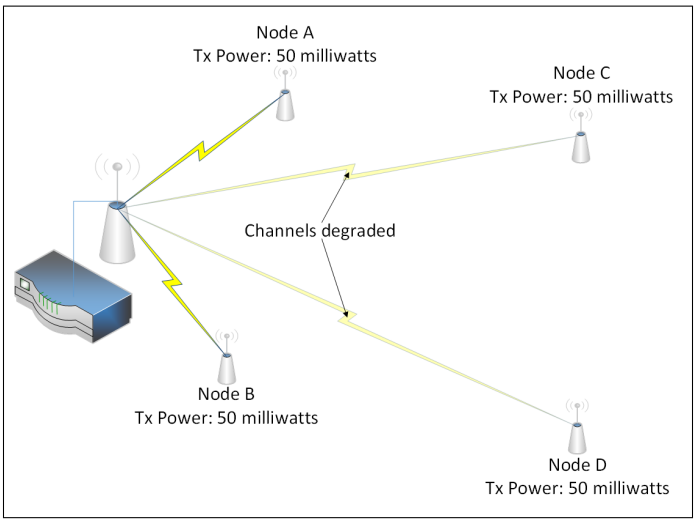


**Figure 1. Initial conditions in cognitive radio network before nodes perform any information exchange, parameter adjustments or algorithm space exploration**

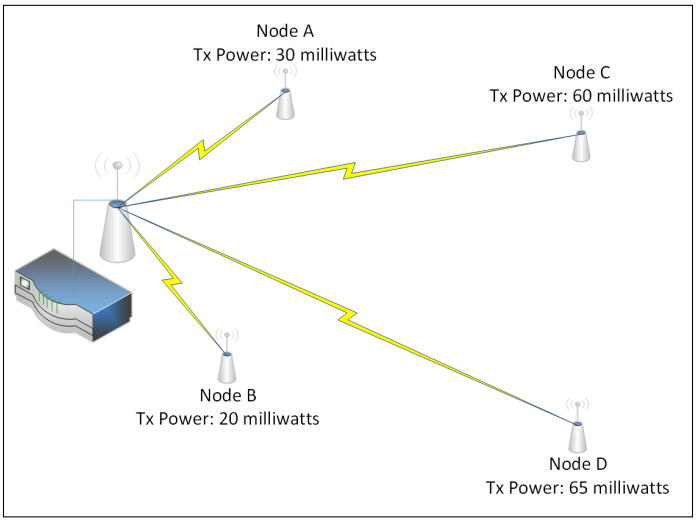


**Figure 2. Cognitive radio network after parameter adjustment**

to inform decisions as to how to alter operating parameters to optimize performance metrics of interest. The scope of information potentially pertinent to the network in these research efforts is thus reduced in comparison to a fully cognitive network as proposed by Mitola. Cognitive networking as originally envisioned is ambitious and a full realization of such a network from the start would be an impossible undertaking. The ongoing research involving cognitive radios and networks is laying the foundation for the full realization of cognitive networking. Two vital components that have emerged from these research efforts are SDRs and Cognitive Engines (CEs).

## 2.3 Software-Defined Radio - The Enabler

Historically, much of a radio's functionality has been implemented in hardware. In such radios, the transmitter chain consists of a series of steps necessary to encode a message onto a carrier signal to be transmitted. Likewise, the receiver chain consists of a series of steps necessary to extract this message from the carrier signal. In a traditional radio, most steps in the transmitter and receiver chain correspond to a separate hardware component that is optimized to efficiently perform the associated signal processing function. As a result, while the efficiency with which the processing required to transmit and receive occurs is fairly high, the radio itself is relatively inflexible in terms of being able to adapt its transmission and processing parameters to its environment. In a traditional radio, such changes would require the radio to be taken offline, dismantled and internal hardware components changed.

In contrast, and as their name implies, SDRs implement much of the traditional radio hardware functionality in software [9]. The line between software and hardware varies from one SDR to another, but on the whole they are much more flexible in terms of varying their transmission parameters and protocols while operating than traditional radios. Rather than having to take the radio offline, the operating pa-

rameters used by the software may be updated to alter the radio's functionality in real-time. As a result, while software-based processing is not quite as efficient as a hardware-based implementation, the increased flexibility offsets the decrease in efficiency. The ability to alter its operation in real-time allows the radio to adapt to its transmission environment on-the-fly. As a result, the SDR is a key enabling technology providing the foundation for fully cognitive networks.

While SDRs are not quite as efficient as traditional hardware-based radios, the literature includes numerous efforts to combine the flexibility of software-based processing and the efficiency of hardware-based computation by moving certain processing tasks back into hardware components such as Field-Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs). One proposed radio receiver architecture uses time-division multiplexing to allow multiple signals to be received and processed using a single receiver chain. The feasibility of the architecture is demonstrated via an FPGA-based implementation. [12] In [13], the authors present a new parallel search-based algorithm for Multiple-Input Multiple-Output (MIMO) detection and demonstrate the algorithm's performance using a GPU. The authors evaluate GPU suitability to serve as a baseband processor for an SDR by exploiting parallelism in the necessary signal processing in [14]. In addition, extensive work to parallelize encoding and decoding for Forward Error Correction (FEC) has been done. A GPU-based Hamming Code decoder is presented in [15]. Further, [16, 17] present work to parallelize the Viterbi decoding algorithm by subdividing computations and recombining the results at the end. Finally, [18, 19] implement linear block decoding on a GPU while [20] uses a GPU for both linear block and convolutional code decoding.

These research efforts may appear counterproductive, moving functionality back into hardware when software-based processing is central to SDRs. That being said,

both FPGA- and GPU-based solutions offer the flexibility of software-defined processing with the speed of hardware-based computation. Neither approach, however, is without its faults. FPGAs can be difficult to learn to use and the development tools available are not as robust as for other applications. Conversely, the development tools available for GPUs are fairly well-developed but they are limited in functionality. Mitola's cognitive networking as originally proposed relied on General-Purpose Processor (GPP)-based SDRs [21]. At least for the time being, however, as GPPs cannot offer the performance necessary to base a fully cognitive node upon them, hybrid solutions combining software- and hardware-based processing are necessary to achieve the performance required for cognitive networking. Increasing the ease of development with FPGAs or reducing the complexity of altering GPU behavior will help improve the suitability of these solutions to the SDR paradigm.

## 2.4 Cognitive Engine - The Brain of a Cognitive Radio

Arguably, the most important component of a cognitive network is the CE. The engine has come to represent the AI component in Mitola's original vision and provides the vehicle by which the network nodes intelligently alter their parameters to optimize network performance [9]. Depending on the architecture in use, the engine could be centrally located or distributed within the network. Importantly, Mitola did not restrict cognitive engines to optimizing network performance by only altering parameters in network nodes that wirelessly transmit. Instead, the engine could control other network components the optimization of which has a positive impact on network performance [9]. It is simply the case that much of the cognitive engine research performed to this point has focused on optimizing the transmission parameters of wirelessly-communicating nodes.

Figure 3, adapted from [21], shows a form of the cognitive cycle normally proposed

for use in cognitive radio nodes. This is the process by which nodes move from sensing their environment and exchanging information to reasoning about changes that can or need to be made and finally to enacting a plan to improve performance. This cycle, the basis for which was laid by John Boyd decades ago, is used to aid decision making in many organizations, including the USAF [22].

In the Observe phase, cognitive nodes sense their environment and form a picture of all relevant operating conditions. Next, in the Orient phase, nodes gather other pertinent information, such as tasks to be performed and the necessary resources. Nodes may also exchange this information with each other or the network infrastructure. In the Plan phase, nodes individually and jointly propose plans to distribute resources (e.g., spectrum) or alter operating parameters to ensure all nodes are able to complete their assigned tasks. Using a pre-defined voting scheme, the nodes then choose a plan to execute in the Decide phase. In the Act phase, the network enacts the chosen plan. In the final phase, arguably the most important, the network gauges the plan's impact on performance.

Depending on the cognitive engine deployed in the network, information about the plan's impact on the network may be stored for later reference. For example, if performance improved, this result and the changes made may be stored in a database. Then, when the network is later trying to form or choose between courses of action, this information may provide insight into what changes may be beneficial. Conversely, if performance degrades, this result and the changes may be stored to prevent the network from trying similar plans in the future. While the Observe and Learn phases are often discussed as the beginning and end of the cycle, respectively, it is important to note there is no real beginning or end to the cycle. Each phase leads to the next, with the Learn phase leading back to the Observe phase. Additionally, not all information exchanges between phases are depicted in Figure 3; most phases have

several outputs that are passed to multiple other phases, not just the one that follows it.

Many CEs using different algorithmic paradigms and parameter sets to optimize various network performance objectives have been proposed in the literature. A two-dimensional chromosome structure for use with different Genetic Algorithms (GA) in tuning multi-carrier network transmission error, power consumption and interference is presented in [23]. In [24], the authors propose a distributed GA for use in optimizing network performance. Importantly, the proposed engine incorporates information from multiple networking layers, referred to as cross-layer analysis or cognition, in its decision-making. As research begins to focus on the improvement in network performance made possible by tuning parameters across multiple layers, the analysis presented in [25] will be important to help inform these investigations. A GA is applied to spectrum allocation amongst cognitive radios, dynamically altering the spectrum allocated to each radio to meet its changing demands while also ensuring secondary transmitters do not interfere with primary transmitter in [26].

Neural networks have also become a popular AI technique on which to base CEs that optimize different aspects of the network in which they are deployed. A neural network-based CE for Access Point (AP) selection in mobile nodes is presented in [27]. The engine is trained on observed transmission conditions and the performance resulting from various AP selections and then helps select which AP should be used as the node moves around. Just as GAs have been used to handle spectrum allocation amongst network users, researchers have also evaluated using neural networks for this purpose. A fuzzy neural network for use in managing spectrum allocation in cognitive networks is presented in [28]. In order to increase the efficiency with which the spectrum must be used, frequency bands that are not in use, referred to as spectrum holes, must first be detected. To this end, [29] presents a neural network

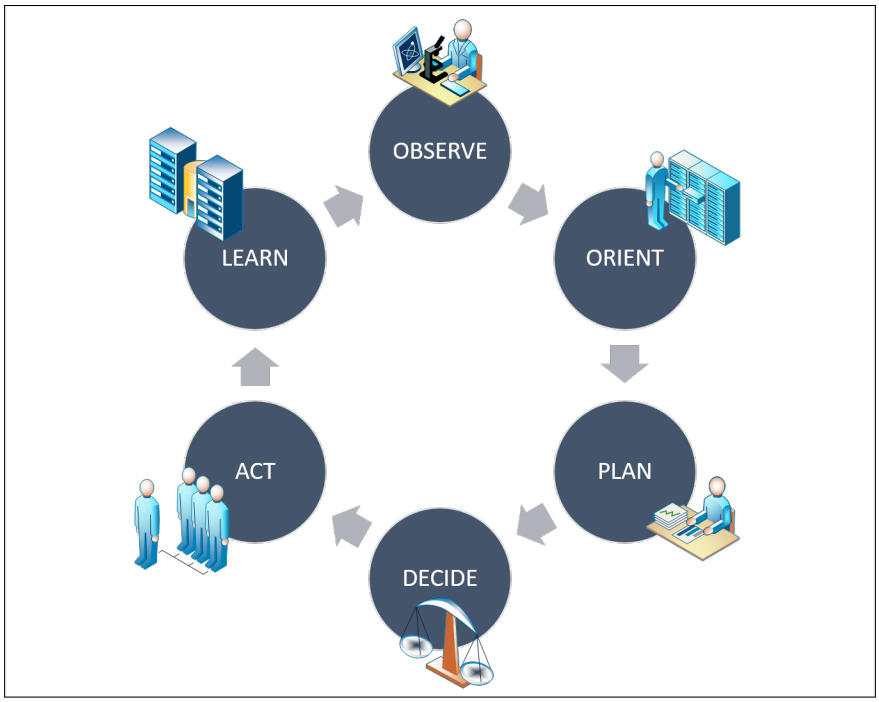


**Figure 3. Cognition cycle used by cognitive nodes in decision making, adapted from [22]**

that, once trained on channel characteristics, may be used to determine whether or not a given band is in use based on currently-observed operating conditions.

There are also numerous proposed CEs that combine techniques or algorithms to create hybrid engines wherein the performance using a combination of approaches is better than what any individual approach offers. A hybrid CE for achieving efficient spectrum management in cognitive networks is detailed in [29]. Determining priority amongst competing SUs is handled by a new queuing scheme and SUs share information with each other as they are added to the network rather than each evolving its behavior in isolation. In [30], a hybrid engine based on Case-Based Reasoning (CBR) and Decision Trees (DTs) is presented. The CBR path in the engine is used when environmental conditions are within a pre-defined bound to a case previously encountered, defined and stored in the engine's database while the engine's DT path is used when operating conditions not previously encountered are observed. In this

case, a new case is defined and stored in the engine's database. A CE based on a hybrid Case-Based Reasoning-Quantum Genetic Algorithm (CBR-QGA) is proposed in [31]. Rather than random population initialization as if often done in GAs, the engine uses case-based reasoning to initialize the population with solutions that are expected to perform well based on previously encountered conditions and the solutions that worked well in those situations. Then, the Quantum Genetic Algorithm (QGA) GA evolves new solutions from the population initialized via CBR. Tests discussed in the paper indicate the new engine outperforms existing CBR-, GA- and QGA-based engines in terms of converge speed and quality of optimization.

Several other AI approaches that have been or could be used as a basis for CEs are explored in [32]. The paper discusses factors that may affect the selection of a given technique for a CE depending on the application and the ramifications the selected AI approach will have on the engine itself.

### 2.5 Cognitive Radio Architectures

Just as many different kinds of CEs have been explored, several architectures have been proposed in the literature for use in organizing cognitive radio nodes and the networks that use them. Some proposals focus more on the theoretical aspects of cognitive node and network architecture. As a result, while useful to a certain extent in the design of real networks, they face several shortcomings that must be addressed in an actual implementation. Other architectures place a greater emphasis on practicality and implementation, arguably providing a more useful foundation on which to build a real radio or network. Regardless for the intended purposes of the two kinds of proposals, both provide important insight and ideas. Perhaps the most important insight is the modularity inherent in the majority of these architectures. The nodes and network are broken down into well-defined components with set interfaces. In

that way, single modules may be changed or updated to alter performance without affecting the rest of the network, as long as the new module conforms to the same interface.

The architecture presented in [33] manages resources using a Cognitive Resource Manager (CRM) to coordinate network reconfiguration actions. The CRM, which is built from several subcomponents, collects pertinent information, makes tuning decisions and maintains a database of current optimization priorities. The CRM's responsibilities closely align with those proposed for the theoretical CE. In [34], an architecture for logically dividing the control plane from the data plane in a cognitive radio network is presented. Nodes interact via the control plane to exchange routing information and constantly maintain optimal routes. Then, when data is to be exchanged, nodes need not first find an optimal route as they are guaranteed by the function of the control plane.

The Iris architecture, developed in [35], is intended for use in organizing the components of a cognitive radio network. While the architecture targets real networks, it is still useful in informing many decisions in designing a simulated network. Similar to the previous architecture, Iris breaks the network and its constituent nodes down into individual components. Different network devices are composed of one or more such subcomponents and the network functionality is implemented through interactions amongst the different devices. Three systems designed and implemented using the Iris architecture, detailed in [36], demonstrate how a wide range of systems may be quickly developed and deployed with the architecture. The first system uses multi-core GPPs to implement a new physical-layer signaling protocol for network coordination. The second system uses the Cell Broadband Engine (CellBE) from the PlayStation 3 to perform cylcostationary signature detection while the third system uses an FPGA to perform energy detection at the receiver to ensure optimal signal

reception. The emphasis on componentization allows existing modules to be reused to simplify and streamline development and deployment.

Two components of interest from this architecture are execution engines and switches. Execution engines (not to be confused with a cognitive engine) define different pathways for data processing within nodes and are composed of multiple subcomponents. For example, execution engines may be used to define the transmit and receive chains within radio nodes. Closely related to the execution engines, switch components are used to choose which data path should be used for processing when multiple paths are available. Thus, the two architectural components may be used to design and organize multi-radio network nodes, with the engines defining transmit and receive chain(s) and the switch choosing which should be used. Another important suggestion set forth by this architecture is to ensure a common interface to all execution engines. In doing so, nodes with arbitrary engines may be built and the engines swapped in and out without affecting the rest of the node. Not only could this paradigm be used to implement devices that operate at the physical layer, it could be used to define submodules that operate at different network layers within a single device.

The Distributed Resource Map (DRM) architecture presented in [37], which targets ad hoc networks with no centralized infrastructure, utilizes a distributed rather than centralized database for storing information. Each node is responsible for sensing the transmission environment and neighbors periodically exchange environmental condition information. The nodes then integrate all of this information together to form a more accurate picture of current operating conditions. In addition to the abstraction components responsible for these tasks, each node has a data acquisition component, a database and a DRM Manager, as shown in Figure 4. Similar to the other architectures, the DRM architecture breaks the network and its nodes

into individual modules that represent discrete units of functionality. By doing so, system capabilities may change or expand by simply changing modules. For example, additional sources of information may be readily utilized by updating the data acquisition component to query the new data source. This modularization allows system capabilities to rapidly change or expand by eliminating the need to update multiple components to implement new functionality.

## 2.6 Intractable Problems and Approximation Algorithms

In order to understand a what a GA is and the need for one in a cognitive engine, two major classes of problems from fundamental computer science theory must first be introduced. At risk of oversimplification, problems for which computer algorithms may be written to find solutions fall into one of two categories. The first, so-called Polynomial-time (P) solvable, are problems for which a computer algorithm may be written to produce provably optimal solutions to instances of the problem and the algorithm's runtime increases according to a polynomial function of the input size. The second major category is the Nondeterministic Polynomial (NP) time set of problems. For these problems, a polynomial-time "certifier," which takes a problem instance and a proposed solution and determines whether or not the proposed solution actually solves the problem instance, may be constructed. However, no algorithm which finds provably-optimal solutions for instances of the problem with a runtime that grows according to a poylnomial of the input's size may be constructed [38]. In general, problems that belong to the class NP are referred to as "intractable" because finding solutions for instances of them becomes computationally infeasible as the input grows beyond a trivially small size. Both the P and NP classes are further broken down into subclasses based on different characteristics, but these distinctions are not germane here.

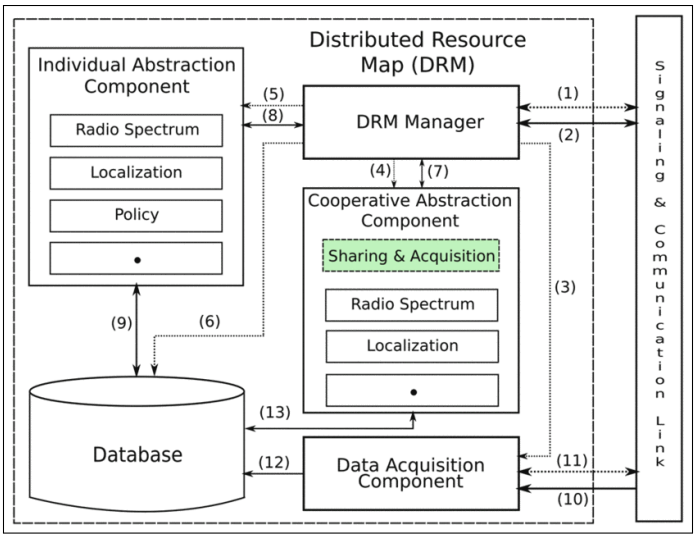


**Figure 4. The Distributed Resource Map Architecture [37], ©[2012] IEEE**

Two different problems from graph theory are used here to illustrate these two types of problems as well as introduce GAs. In general, a graph is a set of nodes $N$ and a set of edges $E$, where each edge $e \in E$ originates at a node $n_1 \in N$ and terminates at another node $n_2 \in N$. Further, a weight $w_n$ may be assigned to each edge $e_n \in E$, where $n \leq |E|$. Graphs are often used to describe some sort of relationship between nodes, with the edges connecting the nodes providing information about the relationship between the nodes it connects. For example, a graph showing American cities with direct flights between them and the cost of each flight is shown in Figure 5. This graph may be used to find the shortest distances between two cities, where "distance" is defined according to the weight assigned to each edge. Thus, as the weights assigned here are flight costs, the graph may be used to find the cheapest flight(s) between two cities. Polynomial-time algorithms, such as Dijkstra's Algorithm, are known that provide provably optimal solutions for this problem [39].

Another fundamental graph theory problem, arguably more interesting because it belongs to the NP class, is the infamous Traveling-Salesman Problem. Put simply,

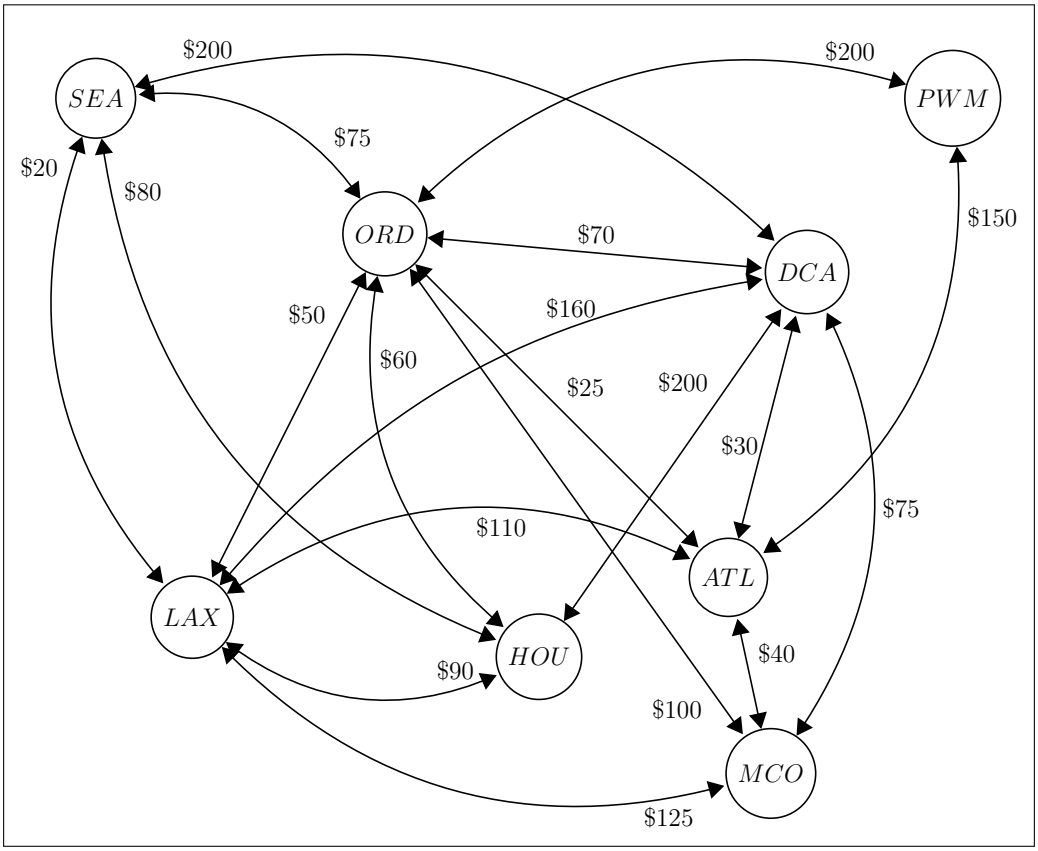


**Figure 5. Example graph showing flights between several American cities with their associated costs**

given a graph, the Traveling Salesman Problem seeks to find the optimal (usually, "shortest" by some metric) route such that the salesman visits all nodes, normally conceptualized as cities, exactly once and returns to the node (city) from which they started [40]. For example, given the graph in Figure 5, the Traveling Salesman may wish to find the least-costly route, starting and ending in Chicago, such that all other cities are visited exactly once.

This may seem trivial at first glance and certainly for a small graph such as the one in Figure 5, finding the optimal solution is relatively easy. However, as the number of edges and nodes in the graph increases, finding the optimal solution becomes more difficult. For example, if the graph included all flights for all airlines between all cities in the United States, finding the optimal solution the Traveling Salesman Problem would be much more difficult. Finding the solution to this problem involves considering all possible paths of length $|N|$ that originate and terminate at a selected node in the graph and visit all other nodes. Then, for each such path, it must be determined whether the path provides a route from the desired starting city to all other cities and then back to the originating city and does so more cheaply than all other routes. In general, there are

$$\frac{(|N|-1)!}{2} \tag{1}$$

such paths. To see this, consider that from the first node there are $|N|-1$ that could be visited next. From the node that is visited first, there are $|N|-2$ nodes that could be visited next. When the last node has been visited, the only node that may be moved to next and still result in a valid solution is the starting node. Thus, there are

$$(|N|-1)\,(|N|-2)\,(|N|-3)\ldots 1 = (|N|-1)!. \tag{2}$$

To obtain the correct number of possible paths and avoid double-counting paths, (2) must be divided by two to obtain the correct number of possible paths [39]. Therefore, in the graph in Figure 5, as the number of flights (edges) grows, the number of possible solutions grows exponentially. As a result, with even a relatively small number of possible flights, the Traveling Salesman Problem becomes intractable.

Rather than simply giving up on NP problems, computer scientists have designed numerous algorithmic paradigms to attempt to solve such problems. Rather than looking to optimally solve instances of these problems, so-called approximation approaches seek to find "good enough" solutions, with some algorithms developed from these techniques being able to guarantee a solution within some bound of the optimal solution. One class of approximation algorithms, to which GAs belong, are referred to bio-inspired because they are modeled after natural phenomena.

As their name implies, the inspiration for GAs comes from cellular mechanisms for heredity and diversity. Within the GA paradigm, a method by which to encode solutions to the problem within a data structure referred to as a chromosome is first developed. In running the algorithm, an initial population is formed according to a pre-defined scheme. Then, across multiple iterations referred to as generations, members from the existing population are selected and a new population member is formed through a process referred to as "crossover" or "recombination" involving the two members selected from the existing population. Before the new member encoding is finalized, it undergoes the "mutation" process in which parts of its encoding are probabilistically altered. Finally, a fitness function is used to assign a value to the new population member in accordance with how well it solves the problem of interest.

The fitness function is arguably one of the most important components of a GA. Members are usually maintained within the population in accordance with their fitness values, and some types of GAs selectively prefer more-fit population members

over less fit members. Thus, the fitness function can have a great effect over which solutions are considered in the search for a "good" solution. A related concept is that of the search space. In general, the search space may be thought of as all possible solutions along with their fitness values, or how well they solve the problem. For problems with only two or three variables, these spaces may be directly visualized by plotting the fitness function. However, as the number of variables begins to grow, visualizing the search space becomes increasingly difficult. However, the search space may still be thought of as having possibly one, but likely more, "good" solutions with high fitness values surrounded by many other less-fit solutions. A GA's recombination mechanism aims to search around known "good" solutions to see if even better solutions may be found in the vicinity (local search) whereas the mutation mechanism seeks to break out of local search to see if better solutions may be found in previously unexplored regions of the search space.

The Traveling Salesman Problem will be used to illustrate the GA paradigm. The first step is to determine how solutions to the problem may be encoded. As solutions consist of different combinations of edges from the edge set, they may be specified as bit strings of length $|E|$. Then, after sequentially numbering the edges as in Figure 6, a 1 in the bit position corresponding to a given edge indicates the edge's inclusion in a proposed solution whereas a 0 indicates the edge is not included in the proposed solution. Given this encoding, the chromosome in Figure 7 encodes a possible solution consisting of edges 1, 2, 4, 10, 11, 16, 17 and 18. That is, the route goes from Portland to Chicago, then to Los Angeles, Seattle, Houston, Washington D.C., Orlando, Atlanta and finally back to Portland at a total cost of $815.

Now that an encoding scheme is established, the next most important part of the algorithm, the fitness function, must be developed. While here the problem itself suggests a fitness function, it may become quite complex depending on the

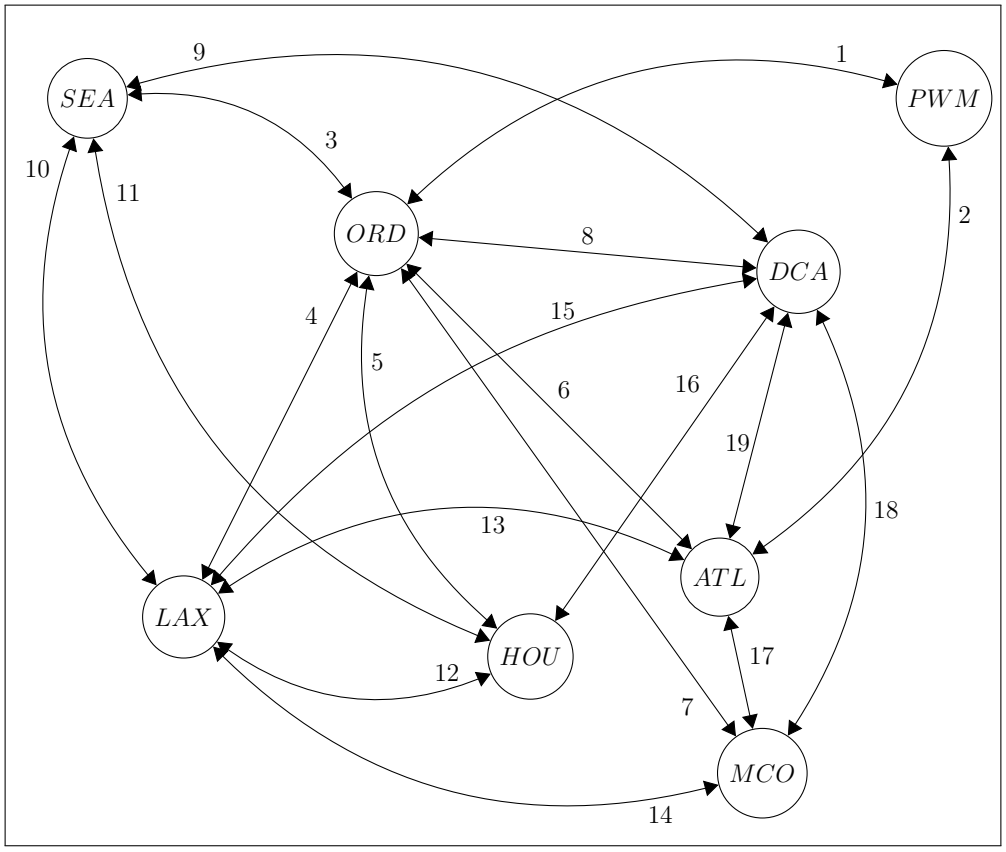


**Figure 6. Example graph showing flights between several American cities with flights (edges) assigned a designated number**

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Chromosome 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| Chromosome 2 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| Bit/Edge Number | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

**Figure 7. Example chromosomes for the Traveling Salesman Problem**

application. For the Traveling Salesman Problem, the fitness function must assess two things. First, it must total the cost of all flights in the proposed solution. Valid solutions may then be compared based on this value to determine which is better. The fitness function must also ensure the proposed solution is valid. That is, it must ensure the route visits all cities exactly once and returns to the starting city. Once the fitness function determines a given solution's value, it assigns that value to the encoded solution.

By comparison, the remaining parts of the GA are relatively straight-forward. The population size (number of visited solutions to track) must be determined as it will affect the portion of the search space that has been explored and is used to generate new solutions. The number of generations to use in developing solutions, which dictates how intensive is the local search around known "good" solutions, must also be determined. Finally, the mutation mechanism, which affects the algorithm's ability to break out of local search and explore other regions of the search space, must be determined. General recommendations as to the population size and number of generations to use are available and several different mutation mechanisms have been proposed [41]. In general, the number of existing solutions selected for recombination, the recombination mechanism itself and the number of new population members thus produced are arbitrary and may be specified by the algorithm's designer, though two solutions are often selected for recombination into a single new solution [42].

Two solutions to the Traveling Salesman Problem given in Figure 6 are used to illustrate the GA's execution. The two solutions used here are shown in Figure 7. The first solution is the same as given previously while the second is another valid solution to the problem. (A randomly-initialized population is assumed here and not included in full; it is further assumed these two solutions were selected at random from this population.) Once the algorithm chooses these two solutions, it must recombine

them in some way to form a new solution. Here, single-point crossover in which the chromosomes are split into two halves and opposite halves recombined to form a new solution, is used. The two solutions are split after the eleventh bit and then the first half of the first chromosome and the second half of the second chromosome are used to form the new population member, given in Figure 8. After the new chromosome is formed via recombination, the new population member must be selectively mutated. Here, each bit is subject to the same probability of mutation of $\frac{1}{19} \cong .053$. Thus, on average, one bit in each chromosome is flipped. Here, the fourteenth bit is randomly flipped and the resulting chromosome is given in Figure 9. After mutation, the fitness function is evaluated on the new population member and, if the new member's fitness value is high enough, it is added to the population. After the new member is selectively added to the population, the entire process begins again with two new members of the population being selected for recombination. This process repeats for however many generations the algorithm is programmed to use in finding "good" solutions. When this number of generations is reached, the algorithm returns the best solution it has found.

Sample pseudocode for a GA is listed in Algorithm 1. As stated previously, several factors such as the population size and number of generations the algorithm is set to use in finding solutions affect its performance. Chapter 3 details the development of the algorithm used in the cognitive engine for the current research.

| New Population Member, Post-Recombination: | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit/Edge Number | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

**Figure 8. New population member after recombination from parent chromosomes, split and recombined after bit 11**

| New Population Member, After Mutation: | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit/Edge Number | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

**Figure 9. New population member after mutation, bit 14 mutated**

**Algorithm 1:** Pseudocode for Generic Genetic Algorithm[42]

**Data:** Number of generations
**Result:** Solution with highest fitness of those considered
Initialize population Evaluate fitness of each individual in initial population
**while** *Number of generations not reached* **do**
  Select $\lambda$ population members for recombination;
  Recombine $\lambda$ members into $\mu$ new population members;
  Mutate $\mu$ new population members;
  Evaluate fitness for each of the $\mu$ new population members;
  Replace the population with the most-fit members from the combined set of original and new members;
**end**

## 2.7 Digital Signal Processing Fundamentals

The concepts of modulation scheme and order, bandwidth and signal power (referred to hereafter as simply "power") are important to the discussion that follows and thus each warrants a brief introduction here. Power is perhaps the easiest of the three to understand and corresponds to the amount of energy with which a signal is transmitted. The power with which a signal is transmitted affects how far it may propagate and still be correctly received. As the signal propagates through the atmosphere, it is attenuated (loses energy) by interactions with molecules in the air. Thus, as the transmitter and receiver move farther apart, all other things being equal, the signal must be transmitted with more power in order to be received.

Power is often reported in units of dBm. This is simply the ratio of the power as given in Watts to 1 milliwatt and may be calculated as

$$P_{dBm} = 10 * \log_{10}\left(\frac{P_w}{1W}\right) + 30 \tag{3}$$

Another important measure related to signal power is signal-to-noise ratio (SNR). As its name implies, this is the average signal to average noise power and is often reported in dB. Similar to dBm, dB is a ratio between two signals, except with dB, the signal to which a signal is compared is arbitrary, not 1 milliwatt [43]. Therefore, as SNR increases, the signal power becomes increasingly stronger than the noise power.

Bandwidth refers to the frequency range across which a given signal is transmitted and relates to the amount of information the signal may carry at a specific point in time. In general, a signal with a wider bandwidth may carry more information. Increasing the bandwidth without similarly adjusting the symbol rate simply leads to more samples for the same symbol rather than additional symbols (information) [43].

FEC may be thought of as adding information to the transmitted signal to combat signal degradation and data loss. With FEC, the transmitter pre-processes the bits to be sent before modulation and adds additional bits to the data. On the receiving end, the receiver uses this additional information to detect and correct bit errors that occurred during transmission. The manner in which the transmitter and receiver process the signal depends on the encoding scheme used and how robust the transmission is against bit errors varies between FEC techniques. Additionally, no FEC scheme can allow for all bit errors regardless of interference level in the transmission medium to be corrected.

Modulation scheme and order refer to the manner in which data to be transmitted is encoded on a carrier signal. In general, carrier signal properties (e.g., frequency, amplitude, phase, etc.) must be altered, or modulated, in such a way that their values, or the difference in their values in successive signal samples, may be interpreted in accordance with an agreed upon scheme and the data extracted. To help illustrate

this, consider a simplistic example in which a signal is modulated between two frequencies. Let the higher frequency stand for a '1' bit and the lower frequency stand for a '0' bit. Given this information, the transmitter may encode the information to be sent and any receivers which know the frequencies (carrier signal properties) to monitor and which frequency corresponds to which bit (modulation scheme) may recover the data. Real-world modulation schemes are much more complex than this simplistic example but it helps illustrate the main idea. Carrier signal properties are altered in a controlled way to encode data and the receivers know the scheme by which the data is encoded and can thus recover the data.

While the modulation scheme in use determines the signal properties that are used to encode data, the modulation order determines the rate at which data may be transmitted [43]. To understand the relationship between constellation size and bits per symbol, the concept of a communications symbol must first be introduced. Rather than transmitting bits, real-world digital transmissions actually transmit these communications symbols. To do this, a constellation that maps the symbols to bit strings must first be constructed. Figure 10 shows a possible constellation for a Phase-Shift Keying (PSK) modulation scheme with a constellation size of 8. Similarly, Figure 11 shows a possible constellation for a Quadrature-Amplitude Modulation (QAM) scheme with a constellation size of 16.

To use these constellations, the bit string each symbol represents must first be determined. Further, to determine how many symbols are needed to transmit $k$ bits in a single symbol, consider the case when $k = 1$. To transmit a single bit at a time, two symbols are needed. One symbol corresponds to '1' and the other symbol to '0'. (In this case, symbols and bits may be thought of synonymously but this does not hold in general.) When $k = 2$, four symbols are needed as there are now four bit strings that could be transmitted (namely, '00', '01', '10', and '11'). For $k = 3$, eight

symbols are needed (the bit strings may be formed by adding a '0' and '1' to the end of the bit strings previously given). Thus, the number of symbols needed to transmit $k$ bits at a time is $2^k$.

While theoretically any size constellation may be used, there is a practical limit in real transmissions. Given the modulation scheme and constellation, the receiver's job is to map the received signal to one of the modulation symbols in order to determine the transmitted bit string. Due to electromagnetic interference (i.e., noise), there is always some uncertainty in the signal. The uncertainty introduced in the signal may be visualized using Figures 10 and 11. In a perfect world, the received symbols would always match the constellation perfectly. In the real-world, however, noise causes the received symbols to be distributed around the constellation point. As a result, the constellation size is constrained by the amount of noise present in the transmission medium and the receiver's sensitivity. If the receiver is not sensitive enough to distinguish between two constellation points, the received signal may be misinterpreted. Similarly, if the interference is high enough, it could cause the receiver to map the signal to the wrong constellation point. In the face of increased noise, then, the constellation size (and thus amount of information that may be sent in one symbol, i.e., data rate) must decrease.

It should be noted the constellation is arbitrary. While the size of constellations used in real-world transmissions are usually a power of two, which symbol corresponds to which bit string and where the symbols are placed on the quadrature and in-phase axes may be adjusted according to application.

## 2.8 Network Simulation Frameworks

There is no shortage of open-source network simulators readily available for use in research by performing high-fidelity simulations of target networks. The upside to

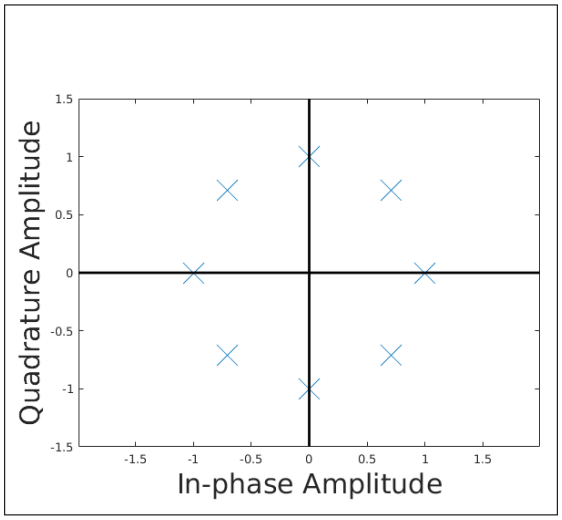


**Figure 10. Example PSK modulation scheme constellation using a modulation order of 8 with each symbol representing 3 bits**

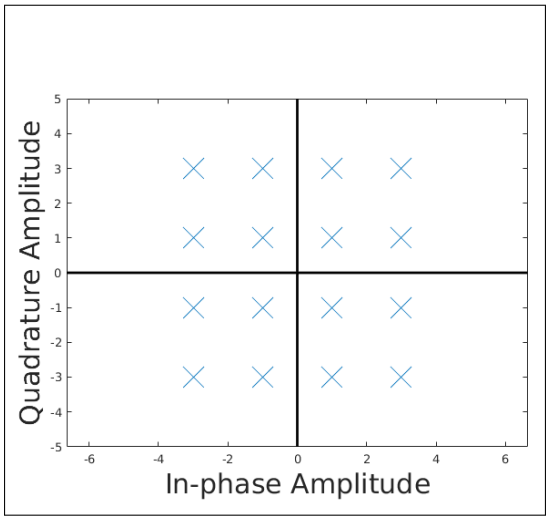


**Figure 11. Example QAM scheme constellation. With a modulation order of 16, each symbol represents 4 bits**

this variety of simulators is it is likely one or more will offer the features necessary for a given simulation. However, the drawback to having several simulation frameworks from which to choose is it can make it more difficult to make a final determination of which framework should be used in a given research effort. There are many factors to consider when choosing a simulation package, including

- Previous experience with one or more simulation frameworks
- The features offered by each framework and those required by the target network simulation
- The languages used in the different frameworks and prior experience with these languages
- Simulation libraries available for the framework and the extent to which necessary functionality is provided in these libraries
- The extent to which the open-source community has embraced the framework. This will affect the quality and quantity of the available supporting libraries as well as the amount of troubleshooting information that may be found on the internet.

Studies that compare different frameworks to each other are available in the literature. Some of the more well-known simulators include Network Simulator 2 (NS-2), Network Simulator 3 (NS-3), the Object Modular Network Testbed in C++ (OMNeT++), SymPy (Python-based simulation framework) and the Java in Simulation Time package [44, 45, 46]. The consensus amongst these studies is that no one simulation framework is the best choice in all situations. As a result, the framework used in a given research effort must be chosen based on a combination of the aforementioned factors. The current research uses OMNeT++ for building and running

the target cognitive network. The reasons for this include prior experience with the simulation package and extensive libraries in OMNeT's INET Framework that may be used or extended to create cognitive networks [47]. Additionally, many in the research community have embraced OMNeT++ as a viable simulation platform for cognitive radios and networks, as evidence by previous research efforts [48, 49, 50].

## 2.9 Virginia Polytechnic Institute and State Uiversity's Cognitive Radio Network

The first framework used to conduct experiments to determine a lower bound on the latency between nodes in a wireless cognitive network is Virginia Polytechnic Institute and State University's (hereafter, Virginia Tech) COgnitive Radio NETwork (CORNET). CORNET is a network of 48-nodes deployed in Kelly Hall on Virginia Tech's campus. CORNET is specifically designed to aid in the development of SDR network protocols by handling the deployment and configuration of the network, freeing CORNET users to focus on protocol development [51]. In order to interact with the network, users first obtain a CORNET account and then remotely connect to the nodes they wish to use in their experiments via Secure SHell (SSH). In order to run experiments, users must define a master scenario controller and a network topology for each scenario. Then, using CORNET's Cognitive Radio Test System (CRTS) software, the user instructs one CORNET node to act as the experiment controller and run the experiment. That node reads the master scenario controller and scenario topology files, initializes the required number of additional CORNET nodes and runs the experiment. The results of the experiment are written to log files in the user's directory, ready to be ingested into MATrix LABratory (MATLAB) and analyzed.

# III. Cognitive Engine and Test Network Design and Implementation

## 3.1 Development of OMNet++ Network for Cognitive Engine Testing

In order to test the engine developed in the next section, a simulated network in which to place the engine is needed. As discussed in Chapter II, OMNeT++ and the INET Framework provide extensive module libraries which implement much of the functionality necessary to build a simulated cognitive network. The network developed here, shown in Figure 12, mainly uses these modules off-the-shelf and only extends a select few in order to integrate the engine into the network. The "networkVisualizer" module visible in Figure 12, is used off-the-shelf and is responsible for helping provide support in visualizing the simulation. (The simulation could be run without visualization, however this could complicate troubleshooting should issues arise.) Simiarly, the "networkConfigurator" and "networkRadioMedium" modules are used unaltered and provide support in configuring the network (for example, setting IP and MAC addresses) and maintaining channel quality information, respectively.

The network nodes visible in Figure 12 are of the "AdhocHost" type, a node type provided by the INET framework. While complete details of the node modules and their submodules are not provided here, a sufficiently detailed discussion in order to illuminate the node components with which the Cognitive Engine (CE) interacts and which therefore need to be altered to allow for this interaction is warranted. (Additional information about components and their properties not discussed here may be found in [47].)

The energy modules, which track the node's energy production, storage and usage, are useful if the simulation should take these factors into account. These subcomponents are not used here but, as Unmanned Aerial Vehicles (UAVs) run on batteries

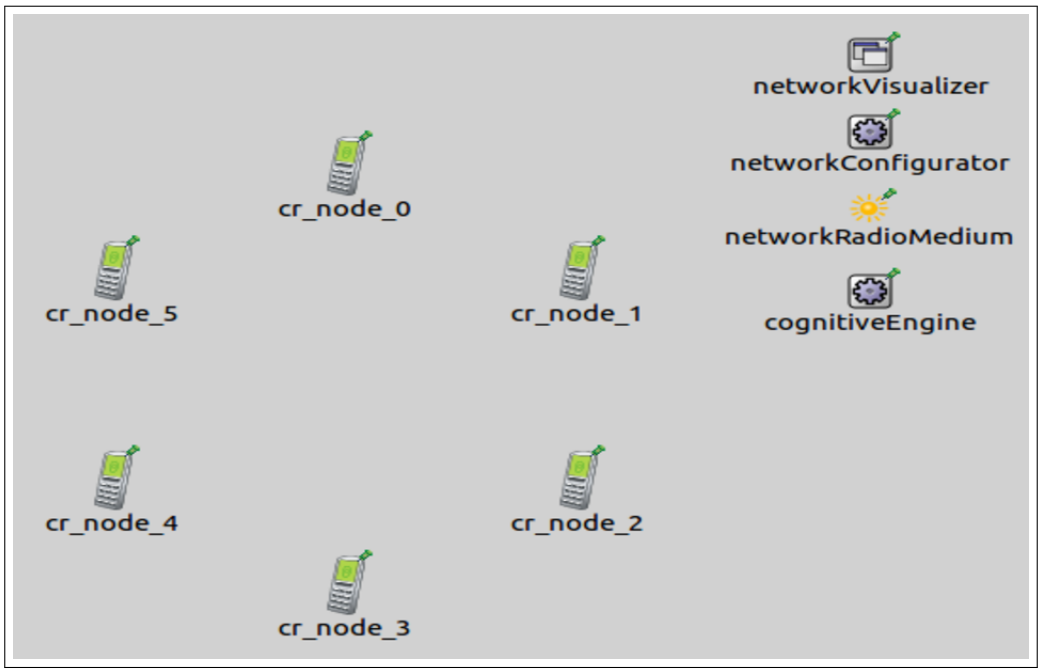


**Figure 12. The OMNeT++ network used to test the cognitive engine**

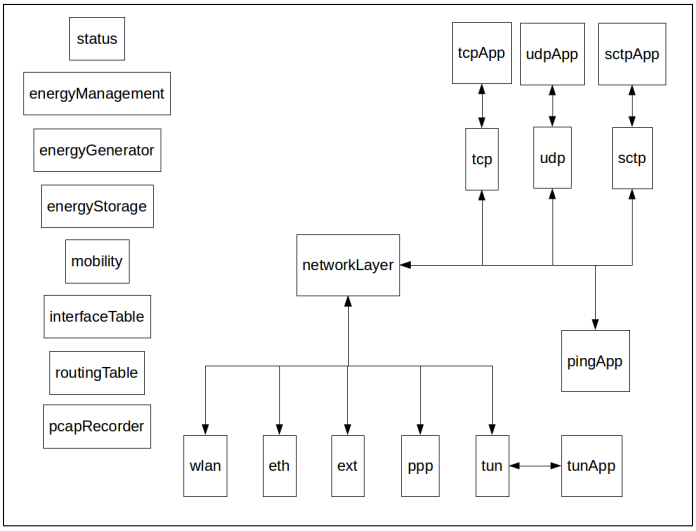


**Figure 13. Internal structure of the "Adhoc Host" module type**

and energy consumption is a concern, could be included in future work to explore how the CE affects battery life. The "mobility" module allows nodes to move in the virtual environment. This component is likewise not used here but could be included in future work to see how the CE performs as the UAVs get closer to and farther away from stationary jammers. The "interfaceTable" and "routingTable" components contain Internet Protocol (IP) and Medium Access Control (MAC) addresses and their associated interfaces, respectively. These components may be manually configured but here, automatic configuration by the network configurator is used. The "pcapRecorder" component may be used if packet captures of messages exchanged between nodes is desired but is not used here.

The Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Stream Control Transmission Protocol (SCTP) applications ("tcpApp", "udpApp" and "sctpApp", respectively) represent other modules which implement one of several different TCP-, UDP- or SCTP-based applications. Custom applications may be implemented by extending one of the existing base applications. As the present experiments are concerned with message latencies rather than their contents, here the "UDPBasicApp" and "UDPSink" modules provided by the INET library are used. The "tcp", "udp" and "sctp" components provide an interface between the node's applications and its network layer ("networkLayer"). The node may optionally also have a ping application ("pingApp"), which talks directly to the network layer, and/or a TUN application ("tunApp"), but these components are not used here.

The last set of node subcomponents ("wlan", "eth", "ext", "ppp" and "tun") provide the external interfaces for messages to leave the node bound for another node. Each component provides an array of the interface type it implements to allow the node to have multiple outbound paths of each interface type. For example, the Ethernet component ("eth" in Figure 13) could be used to provide multiple ports on

a simulated router, analogous to the physical ports on a real router. Here, only the "wlan" component is used to provide a wireless interface between the node and its transmission environment.

The "wlan" node component provides an array of wireless network interface cards (NICs). The INET framework provides several working NICs and additional types, which extend the functionality of the existing types, may be implemented. Here, the "WiressNIC" module provided by the INET framework is used. Figure 14 shows the internal structure of this module type. The "classifier" component provides quality-of-service (QoS) classification for incoming messages. The "mac" component provides the implementation of the MAC protocol that is to be used for incoming and outgoing messages. Here, the standard Carrier-Sense Multiple Access with Collision Avoidance (CSMA-CA) protocol, provided by the INET framework, is used. Additional MAC protocols may be implemented and used by substituting them in for this component. Finally, the "radio" component provides the implementation of the radio used for transmitting and receiving radio signals. Here, the "APSKScalarRadio" module type provided by the INET framework is used. The INET framework includes additional radio types and new radios may be implemented by extending the existing types.

Figure 15 shows the internal structure of the "CRCENNetworkRadio", a new radio type, implemented by extending the provided "APSKScalarRadio" module type in order to allow the radio to process incoming messages from the CE and update its operating parameters. When the radio receives a message, it first checks the message kind to determine if the message originated at the CE. If so, the radio further processes the message to extract the encoded parameters and updates its operating characteristics accordingly. If the message did not come from the CE, the radio processes it in the same manner as the base "APSKScalarRadio" module type. The code for the "CRCENetworkRadio" is given in Appendix B.

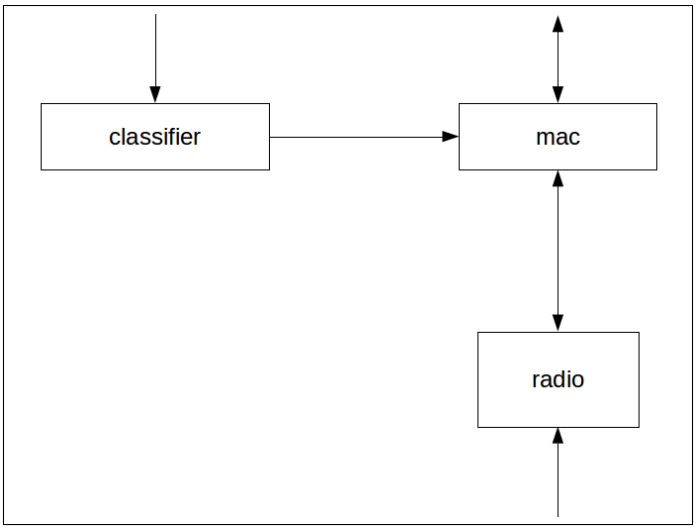


**Figure 14. Internal structure of the "Wireless Network Interface Card" module type**

In Figure 15, the "energyConsumer" component has a similar function as in the "AdhocHost" module; it aids in tracking the radio's energy usage and is not used here. The "transmitter" and "receiver" implement the radio's transmit and receive signal processing chains, respectively. Finally, the "antenna" component implements the radio's antenna and interacts with the network's radio medium to determine transmission range and which other nodes will receive a given transmission. The INET framework provides several transmitter and receiver chains as well as several antenna types. Here, the "IsotropicAntenna", "APSKScalarTransmitter" and "APSKScalarReceiver" module types, provided by the INET framework, are used.

## 3.2 Cognitive Engine for Optimizing Latency

### Introduction.

As discussed when Chapter 2 introduced the concept of a cognitive radio, the CE is the radio's brain. In developing a CE, the main characteristics that distinguish one engine from another is the Artificial Intelligence (AI) technique upon which the

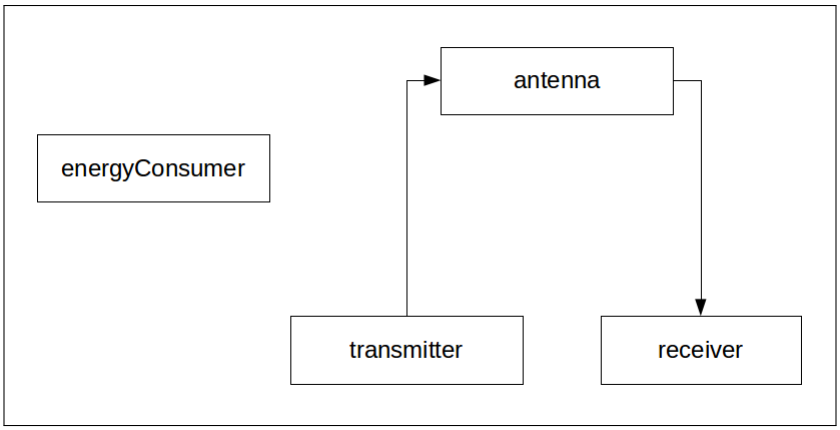


**Figure 15. Internal structure of the new radio type, extended from the "APSK Scalar Radio" module type, to allow the engine to update the radio's operating parameters**

engine is based and the manner in which the engine relates the tunable parameters to the performance objectives. In light of [24], which provides several reasons why heuristic-based CEs may be preferable to other AI approaches such as expert systems or neural networks, the CE developed here is based on a genetic algorithm. In comparison to genetic algorithms and other heuristic AI techniques, expert systems, neural networks and similar AI approaches are relatively inflexible in the face of previously unencountered situations and may fail to find acceptable solutions in such cases. In heuristic-based approaches, such as the genetic algorithm-based engine developed here, a fitness function relates the tunable parameters to the performance objectives. This function accepts information regarding the transmission environment and a specific assignment of values to each of the tunable parameters and outputs a fitness value based on this information. This value measures how suitable is a given solution and is used to compare one solution to another. Importantly, just as the CE forms the core of a cognitive radio, the fitness function forms the core of heuristic-based CEs. Two different engines could use the same tunable parameter and performance objective sets but use different fitness functions and, as a result, output different solutions.

**Deciding Performance Objectives and Tunable Parameters.**

Closely related to the fitness function is the choice of tunable parameters and optimization objectives. Determining the performance characteristics the engine should optimize is relatively straight-forward. As stated previously, latency is the main performance objective of interest here. The CE should operate to minimize the delay between when a transmitter sends a message and the intended receiver receives the messages. While there is no requirement a fitness function or CE take more than one objective into account, care should be taken to try and anticipate whether a single-objective function could lead to solutions that optimize the fitness function's value but are clearly undesirable. As a simplistic example, one way to optimize latency would be to prevent any nodes from sending messages, resulting in zero latency. While this would optimize a fitness function that only takes latency into account, such a solution is clearly undesirable; the network is useless if nodes cannot communicate. Without additional objectives in the fitness function, however, the CE has no way of knowing the optimal solution it has decided upon is not acceptable. As a result, the CE developed here will also take throughput into account in optimizing the network's performance. In attempting to reduce latency, the goal of simultaneously maximizing throughput should drive the CE away from solutions that reduce latency by simply reducing traffic. Finally, as the goal of this research is to explore the feasibility of incorporating a real-time cognitive engine into a UAV swarm, power consumption is also a performance metric of interest. While power is not included in the simulation itself, incorporating power considerations into the fitness fuction will drive the CE away from solutions that solve latency issues by simply increasing the power output of transmitters. Therefore, the fitness function developed for the CE will be a multi-objective function, though it will give greater weight to the latency objective than to the throughput or power objectives.

Compared with deciding upon the set of performance objectives, determining which parameters to allow the CE to use in tuning the network's performance is more challenging. Based on their use in the preliminary latency experiments detailed in Section 4.1, the tunable parameter set here includes both modulation scheme and order. While a fitness function could be based solely on these two parameters, this would result in a trivial search space and negates the power of a genetic algorithm. The advantage of biologically-inspired algorithms such as a genetic algorithm is their ability to efficiently explore large search spaces and find good (though not guaranteed optimal) solutions. As a result, the tunable parameter set should include additional variables to allow for a larger number of possible solutions.

The challenge, then, is to determine which additional parameters to include in the tunable parameter set to allow for a greater number of possible solutions. Parameters that have a greater impact on system performance are clearly preferable to parameters that have a smaller performance impact. [52] surveys several performance parameters and categorizes their impact on system performance with respect to a combination of five different goals. Consistently, of the parameters considered and not already in the tunable parameter set here, power and bandwidth showed the greatest impact on performance. The survey emphasized minimizing bit error rate (BER) while also considering throughput. While these are not the same objectives being sought here, it is reasonable to expect that parameters that have a considerable effect on BER or throughput will have a similar effect on latency. For example, as used here, latency measures the time between when a message is sent from the network layer at the sender and when the network layer in the receiver receives the message. This potentially requires the sender to prepare multiple messages at the link and physical layers and then transmit these messages to the receiver. A delay in any of these messages will increase the latency in receiving the message at the

network layer in the receiver. As a result, increasing BER, which could potentially cause a greater number of retransmissions at the link layer, would increase latency. Therefore, the tunable parameter set also includes power and bandwidth.

The final step in determining the set of tunable parameters is deciding on valid values for each variable. While a large search space is desirable, it is possible for the search space to grow too large. In such cases, a genetic algorithm would either need an inordinate amount of time to search a reasonably-sized portion of the search space to increase confidence it finds a good solution or it would search a portion of the search space that is too small and which yields low confidence it finds a reasonably good solution. Constraining the modulation schemes under consideration is relatively easy. The INET modules in use here only support Quadrature Amplitude Modulation (QAM) and Phase Shift Keying (PSK) modulation. For constellation size, the INET modules in use allow for powers of two up to and including 256. Such a large constellation size is virtually impossible to achieve with PSK modulation and difficult to achieve with QAM except in scenarios with extremely low noise levels. The allowed values for power and bandwidth are adapted from [53], with the minimum and maximum values and ranges slightly altered to result in a number of possible values that is a power of two. This is to ensure all possible bit strings for the genes encoding each parameter are valid. The chromosome structure for the genetic algorithm is presented in the next section. Table 1 summarizes the tunable parameters, their maximum and minimum values and their step size.

**Table 1. Parameters available to genetic algorithm for use in tuning network performance, along with minimum and maximum values and step size**

| Parameter | Minimum Value | Maximum Value | Step Size |
|---|---|---|---|
| Modulation Scheme* | NA | NA | NA |
| Modulation Order | 2 | 256 | Powers of two |
| Power (dBm) | 4 | 35 | 1 dBm |
| Bandwidth (mHz) | 1 | 32 | 1 mHz |
| * - Allowable values are PSK and QAM | | | |

### Chromosome Structure.

Before developing the fitness function, Table 1 may be used to determine an appropriate chromosome structure to encode the parameter settings. First, as there are only two possible values for the modulation scheme, a single bit may be used to encode this value, where "0" corresponds to PSK modulation and "1" encodes QAM. Next, there are 8 possible values for the modulation order, requiring 4 bits for encoding. Care must be taken during the genetic algorithm's execution to ensure no invalid encodings are generated either through random population initialization or mutation. For example, "0000", denoting a modulation order of zero is invalid, as is anything greater than "1000". (Theoretically, constellation sizes larger than 256 are possible, but are rarely seen in practice and thus are not considered here.) For power, there are 32 possible values, requiring 5 bits to encode each value. Due to the range of values considered, "00000" corresponds to a power setting of 4 dBm and "11111" corresponds to a value of 35 dBm. Therefore, 4 must be subtracted from the desired power setting before encoding and added to the decoded value to obtain the correct power setting. Similar to power, there are 32 possible values for bandwidth, also requiring 5 bits to encode each value. Due to the range of values considered, "00000" corresponds to a bandwidth of 1 MHz while "11111" denotes a value of 32

MHz. Therefore, 1 must be subtracted from the desired bandwidth setting before encoding and added to the decoded value to obtain the correct bandwidth setting.

Figure 16 shows an example chromosome containing a set of parameter values. Applying the encoding outlined above, this chromosome encodes a modulation scheme setting of QAM. Further, the bits per symbol setting contained within the chromosome is two while the power setting is 11 (from chromosome) +4 (required offset) = 15 dBm. Finally, the bandwidth setting is 13 (from chromosome) +1 (required offset) = 14 MHz.

**Relating the Tunable Parameters to the Performance Objectives through a Fitness Function.**

Now that the performance objectives of interest and parameters the engine may use to tune network performance are established, the function that will be used to produce a fitness value for each solution considered must be developed. Rather than developing a single function to capture all performance objectives in a multi-objective problem, it is usually easier to develop separate fitness functions for each objective and then combine them together. This approach will be used here. In order to accomplish this, care must be taken to ensure the possible range of values output by each of the individual fitness functions is the same. This is often accomplished by normalizing the

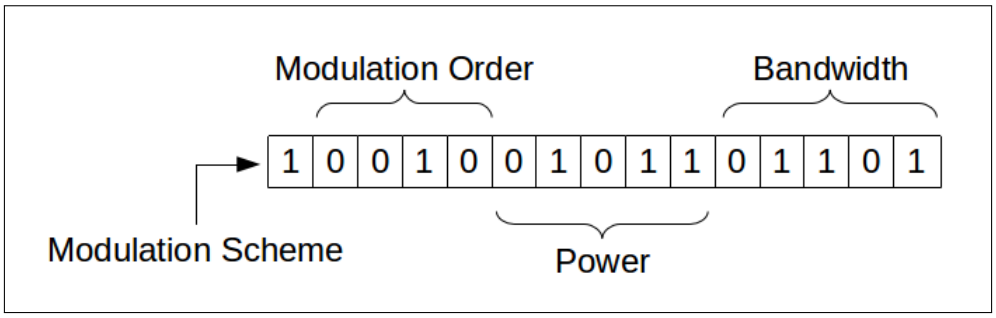


**Figure 16. Example of parameter encoding in chromosome**

functions to output a value between 0 and 1. Finally, it is also easier if all objectives have the same goal (i.e., minimization or maximization). Due to the manner in which the individual objectives are normalized here, discussed below, latency and throughput fitness value close to 1 are preferable while power fitness values close to 0 are preferable. Before the power fitness may be added to the fitness value for the other two objectives, it must be transformed into a maximization problem. Otherwise, tradeoffs in latency or throughput and power consumption may mask which of two solutions is preferable. Thankfully, as the possible values that may be returned by each of the fitness functions is normalized to the range of 0 to 1, this transformation may be easily accomplished by simply subtracting the fitness value returned for the power objective from 1. In that way, the goal of minimizing power consumption is the same as maximizing the resulting value when the normalized power fitness value is subtracted from 1. With the fitness function structure as described, solutions with larger fitness values are preferable and different performance objectives may be given different levels of importance by incorporating weights into the fitness function.

**Latency Fitness Function.**

The latency fitness function is developed in a similar manner as the throughput fitness function in [53], by taking into account the probability of a bit error based upon the transmission parameters in use and the channel quality. An expression for the latency in terms of the physical layers frames required to transmit the network layer packet, the probability a given frame is received in error and the latency for a single frame is first developed. This expression is then normalized to produce a result between 0 and 1, the associated latency fitness. Let $PS$ denote the packet size of a network level packet and let $PS_H$ denote the size of the header associated with the packet. Further, let $DG$ denote the length of a link layer datagram and $DG_H$

denote the datagram header size. Finally, let $FS$ denote the length of a physical layer frame and let $FS_H$ denote the frame header size. To determine the total number of frames that must be transmitted, the total number of bits that must be transmitted must first be calculated. This may be determined by adding the required number of datagram header bits to the packet data and header bits and dividing by the frame size. Therefore, the number of frames that must be transmitted to send a single network level packet is

$$\#\text{ Frames} = \left\lceil \frac{PS + PS_H + \frac{PS+PS_H}{DG} * DG_H}{FS} \right\rceil. \tag{4}$$

Now consider a single physical layer frame and let $P_{be}$ be the probability a single bit is received in error at the receiver. In order for a single frame to be received successfully, all of the $(FS+FS_H)*8$ bits that compose the frame must be successfully received. The probability of the receiver receiving a single bit correctly is

$$P_{\text{Bit successfully received}} = 1 - P_{be}. \tag{5}$$

Therefore, for the frame to be successfully received, this event must occur $(FS + FS_H) * 8$ times in a row. The probability of a frame being successfully received is

$$P_{\text{Frame successfully received}} = (1 - P_{be})^{(FS+FS_H)*8}. \tag{6}$$

Conversely, the probability of a frame not being received successfully may be given as

$$\begin{aligned} P_{\text{Frame not successfully received}} &= 1 - P_{\text{Frame successfully received}} \\ &= 1 - \left( (1 - P_{be})^{(FS+FS_H)*8} \right). \end{aligned} \tag{7}$$

Latency for a single network layer packet may be thought of as the time it takes

to transmit a single frame from the sender to the receiver multiplied by the expected number of frames required to transmit the packet from the sender to the receiver, when bit errors are taken into consideration. In order to determine an expected number of frames that must be transmitted to successfully send the network packet, a Geometric distribution [54] using the probability of a frame not being successfully received as the probability of failure may be set up as

$$f(x) = (P_{\text{Frame not successfully received}})^{x-1} * (1 - P_{\text{Frame not successfully received}}), \tag{8}$$

where x is a geometrically distributed random variable denoting the expected number of times a single frame must be sent before it is successfully received. Therefore, the number of frames that must be sent to transmit a given network layer packet from the sender to the receiver is

$$E[\text{Frames}] = E[x] * (\#\ \text{Frames}), \tag{9}$$

where $E[x]$, the expected number of times a given frame must be sent before it is received correctly, may be calculated from general results for the Geometric distribution [54] as

$$E[x] = \frac{1}{1 - P_{\text{Frame not successfully received}}}. \tag{10}$$

Finally, to calculate latency, the expected number of frame transmissions is multiplied by the per-frame latency, denoted $Latency_F$. Thus

$$
\begin{aligned}
\text{Latency} &= E[\text{Frames}] * \text{Latency}_F \\
&= E[x] * (\#\ \text{Frames}) * \text{Latency}_F \\
&= \frac{1}{1 - P_{\text{Frame not successfully received}}} * \left\lceil \frac{PS + PS_H + \frac{PS+PS_H}{DG} * DG_H}{FS} \right\rceil * \text{Latency}_F \\
&= \frac{1}{1 - \left(1 - \left((1 - P_{be})^{(FS+FS_H)*8}\right)\right)} * \left\lceil \frac{PS + PS_H + \frac{PS+PS_H}{DG} * DG_H}{FS} \right\rceil * \text{Latency}_F.
\end{aligned}
\tag{11}
$$

While (11) gives an expression for the latency, as discussed previously, this value must be normalized to a value within the range of zero to one so it may be combined with the other fitness function values to yield an overall fitness value for a given parameter set. To do this, the expression given in (11) may be compared to the latency that would result if there were no bit errors. In this case, (11) reduces to

$$
\begin{aligned}
\text{Latency}_{\text{No bit errors}} &= \frac{1}{1 - \left(1 - \left((1 - 0)^{(FS+FS_H)*8}\right)\right)} * \left\lceil \frac{PS + PS_H + \frac{PS+PS_H}{DG} * DG_H}{FS} \right\rceil * \text{Latency}_F \\
&= \frac{1}{1 - \left(1 - \left((1)^{(FS+FS_H)*8}\right)\right)} * \left\lceil \frac{PS + PS_H + \frac{PS+PS_H}{DG} * DG_H}{FS} \right\rceil * \text{Latency}_F \\
&= \frac{1}{1 - (1 - 1)} * \left\lceil \frac{PS + PS_H + \frac{PS+PS_H}{DG} * DG_H}{FS} \right\rceil * \text{Latency}_F \\
&= \left\lceil \frac{PS + PS_H + \frac{PS+PS_H}{DG} * DG_H}{FS} \right\rceil * \text{Latency}_F.
\end{aligned}
\tag{12}
$$

Therefore, to normalize the latency value, the result of (12) may be divided by the result of (11), yielding

$$\begin{aligned}\text{Latency}_{\text{Normalized}} &= \frac{\left\lceil \frac{PS+PS_H+\frac{PS+PS_H}{DG}*DG_H}{FS} \right\rceil * \text{Latency}_F}{\frac{1}{1-\left(1-\left((1-P_{be})^{(FS+FS_H)*8}\right)\right)} * \left\lceil \frac{PS+PS_H+\frac{PS+PS_H}{DG}*DG_H}{FS} \right\rceil * \text{Latency}_F} \\ &= \frac{1}{\frac{1}{1-\left(1-\left((1-P_{be})^{(FS+FS_H)*8}\right)\right)}} \\ &= 1-\left(1-\left((1-P_{be})^{(FS+FS_H)*8}\right)\right). \end{aligned} \tag{13}$$

In order to evaluate the latency fitness function (13), an expression for bit error probability is needed. Such an expression is given in theoretical results for different modulation schemes, reproduced below [55].

$$P_{PSK,BER} = \frac{2}{\log_2 M} * Q\left[\sqrt{T_0 B}\sin\left(\frac{\pi}{M}\right)\sqrt{r\frac{2C}{N}}\right] \tag{14}$$

Similarly, the result for QAM is

$$P_{QAM,BER} = \frac{4}{\log_2 M} * \frac{\sqrt{M}-1}{\sqrt{M}} * Q\left[\sqrt{\frac{3rT_0BC}{(M-1)N}}\right]. \tag{15}$$

Table 2 details the meaning of each symbol in the above equations.

**Table 2. Meaning of symbols in the equations for BER for PSK and QAM modulation schemes**

| Symbol | Meaning |
|---|---|
| $T_0$ | Symbol Period |
| B | Bandwidth |
| C | Carrier Power |
| N | Noise Power |
| M | Modulation Order |
| r | Coding Rate Factor |

Further, the $Q$ that appears in (14) and (15) is a reference to the Q function, which is defined as

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{x^2}{2}} dx. \tag{16}$$

The bandwidth, carrier power and modulation order are three of the variables the CE has at its disposal to optimize network performance so their values will be known when trying to evaluate the fitness function. Similarly, the noise power may be obtained from the radio medium module within the simulation. The symbol period is a function of the bitrate, which is set at the beginning of the simulation and is not allowed to vary, and the modulation order. In particular, dividing the bitrate by the modulation order gives the symbol rate. Then, taking the inverse of this value yields the symbol period. Thus, all the necessary values are known when trying to evaluate the fitness function.

The more difficult step in determining the bit error rate when trying to calculate fitness values is the evaluation of the Q function. The integral in Equation 16 has no closed-formed solution. While it could be numerically evaluated, the usual approach is to use a pre-compiled table, similar to [56], to look up the approximate value. This is the approach used here.

**Throughput and Power Consumption Fitness Functions.**

Unlike latency, fitness functions that relate throughput and power consumption to network operating parameters are widely available. Here, fitness functions originally given in [53] for these objectives, reproduced below, will be used.

$$\text{Throughput}_{\text{Normalized}} = \frac{FS}{FS + FS_H + DG_H + PS_H} * (1 - P_{be})^{FS+FS_H} * R_c * TDD, \tag{17}$$

where $FS$, $FS_H$, $DG_H$, $PS_H$ and $P_{be}$ are as previously defined, $R_c$ is the bitrate when coding is taken into account and $TDD$ takes time-division duplexing into account. The network simulated here does not use Forward Error Correction (FEC) coding nor does it implement time-division duplexing. Thus, $R_c$ and $TDD$ are both set to 1. The fitness function for power consumption is

$$\begin{aligned} \text{Power Consumption}_{\text{Normalized}} = \alpha * & \frac{(P_{max} + B_{max}) - (P + B)}{P_{max} + B_{max}} \\ + \beta * & \frac{\log_2(m_{max}) - \log_2(m)}{\log_2(m_{max})} \\ + \lambda * & \frac{R_{smax} - R_s}{R_{smax}}, \end{aligned} \tag{18}$$

where $P_{max}$ and $B_{max}$ are the maximum transmit power and bandwidth, respectively, $P$ and $B$ are the actual transmit power and bandwidth being used, $m_{max}$ is the maximum possible modulation order and $m$ is the modulation order in use, $R_{smax}$ is the maximum symbol rate and $R_s$ is the symbol rate in use. The first component takes into account the transmit power and bandwidth. The second two terms take into account increased processing, and thus increased power consumption, required when using progressively larger modulation orders and a larger sample rate. $\alpha$, $\beta$ and $\lambda$ are weighting parameters used to give the three components of the power consumption fitness function different levels of importance.

**Complete Fitness Function.**

Taking the preceding into account, the complete fitness function is given as

$$
\begin{aligned}
F\left(\text{Modulation}, M, C, B\right) = W_1 * \left[1 - \left(1 - \left((1 - P_{be})^{(FS+FS_H)*8}\right)\right)\right] \\
+ W_2 * \left[\frac{DG}{DG + DG_H + F_H} * (1 - P_{be})^{DG+DG_H} * R_c * TDD\right] \\
+ W_3 * \left[\alpha * \frac{(P_{max} + B_{max}) - (P + B)}{P_{max} + B_{max}}\right. \\
+ \beta * \frac{\log_2(m_{max}) - \log_2(m)}{\log_2(m_{max})} \\
\left. + \lambda * \frac{R_{smax} - R_s}{R_{smax}}\right].
\end{aligned}
\tag{19}
$$

From Equation 19, it may be seen that for the latency and throughput objectives, lower BERs correspond to more perfect reception that closely matches the theoretical performance achieved in the absence of bit errors. As this is the value by which the realized latency and throughput are normalized, fitness values close to 1 indicate performance that more closely matches this limit while fitness values close to 0 indicate performance that is considerably below this upper bound. Thus, latency and throughput fitness values close to 1 are preferable. Similarly, for power fitness, before subtraction from 1, values close to 1 indicate power consumption that is approaching the maximum possible power consumption while values close to 0 indicate little power consumption. Thus, maximizing the value that is obtained when the power fitness value is subtracted from 1 is the same as minimizing the power fitness value and, by extension, the power consumption.

**Cognitive Engine Algorithm.**

Before concluding the chapter, the engine's algorithm, pseudocode for which is given in Algorithm 2, is briefly introduced. (The full code for the engine is given in Appendix B.) As mentioned when they were introduced in Chapter II, a genetic algorithm's performance is largely dependent upon the population size and the number

of generations used. Here, a baseline population size of 50 and number of generations of 200 is used. However, as detailed in Chapter IV, the effect of both the population size and number of generations the engine uses in generating new solutions on the fitness of found solutions and the engine's runtime are explored.

From Algorithm 2, the required inputs for the engine's genetic algorithm include the current population and the number of generations the engine should use in producing new solutions. Once the engine produces the number of specified generations, it returns the most-fit solution it has found. In generating a new population member, the engine randomly selects two existing population members. The engine then splits these members into their constituent genes. For each of the new population member's genes, the engine randomly selects one of the two genes contributed by the parent population members and assigns it to the new population member. Once the engine forms the new population member, it mutates the newly formed chromosome. For modulation scheme, power and bandwidth, this mutation is done randomly such that, on average, one bit in each chromosome is mutated. Due to restrictions on the modulation order, mutating this gene is not handled in the same fashion. First, if the mutated modulation scheme is QAM, the mutated modulation order must be even. Further, regardless of the mutated modulation scheme, the mutated order cannot exceed 8 (a binary string consisting of a 1 followed by three 0s). Therefore, if the mutated first bit is a 1, it sets all other bits. Thus, the mutation is handled such that, on average, the bit only mutates one out of every eight times. The remaining bits are mutated in a similar fashion as the bits for the other genes. Once the engine mutates the modulation order, a final check to ensure QAM and an odd modulation order are not combined in the new population member. If the engine detects this situation, it sets the final bit of the modulation order gene to 0 and randomly selects one of the other 3 bits to set to 1.

**Algorithm 2:** Pseudocode for Cognitive Engine Algorithm

```
Data: Current Population, number of generations
Result: Most-fit population member
while Number of generations not reached do
    Randomly select two population members;
    Split members into component genes;
    for Each gene in new population member do
        Randomly select gene from one of the two parents
    end
    Mutate new population member Calculate fitness for new population
     member;
    Add new member to population;
end
```

# IV. Research Methodology

## 4.1 Latency Experiments using Virginia Tech's CORNET

In exploring a cognitive network's performance with a genetic algorithm-based engine that targets latency for optimization, a lower bound on the latency of exchanged messages in a wireless network to which the cognitive network's performance may be measured is needed. These initial tests will be conducted using Virginia Tech's CORNET which was briefly described in Chapter II. Though these tests could be run with a set configuration (e.g., a single modulation scheme, constellation size and data rate), the results may only be valid for that particular configuration. It is expected latency will be largely independent of such factors as the major contributors to latency are processing steps that must occur in all transmitters and receivers. However, to ensure no unanticipated factors cause the latency lower bound determined through this experiment to fluctuate significantly across configurations, several combinations of modulation scheme, constellation size and data rate will be tested.

**Independent and Dependent Variables.**

One of the most powerful aspects of the CORNET framework is the considerable number of transmission parameters it allows users to vary during the course of their experiments. The Cognitive Radio Test System (CRTS) allows the user to specify over 50 different transmission parameters in a given scenario, including transmitter and receiver modulation scheme, constellation size, data rate, frequency, gain, subcarriers and Forward Error Correction (FEC). Additionally, a user-provided Cognitive Engine (CE) may alter any of these parameters as deemed necessary during the scenario. As the number of parameters that may be altered, as well as the number of values each parameter may take on, lead to a number of test cases that exhibits exponential

growth, the first step in the design of the CORNET experiments is to identify a subset of the parameters to be investigated (independent variables) and the resulting system response indicators (dependent variables) of interest. As previously identified, the network performance metric (dependent variable) of interest here is the latency with which network-level packets are sent and received between nodes.

The independent variables under investigation are modulation scheme, constellation size and data rate. Within CRTS, the modulation scheme and bits per symbol are controlled by the same parameter. The modulation scheme affects the efficiency with which data is encoded and thus the rate at which data is transmitted from sender to receiver. It is expected latency with Quadrature Amplitude Modulation (QAM) will be less than with Ampitude Shift Keying (ASK), which is expected to be less than the latency with Phase Shift Keying (PSK) due to a decrease in the encoding efficiencies across the three modulation schemes. Further, it is expected that as the constellation size used in conjunction with a given modulation scheme increases, the latency will decrease. As discussed in Chapter 2, as the constellation size increases, the number of bits that may be sent simultaneously also increases. Therefore, with a larger constellation size, information may be sent at a faster rate and the time required for the sender to transmit a given packet to the receiver is less than the time required with a smaller constellation size. Finally, it is expected the data rate will likely increase latency. As the data rate increases, the radios are exchanging messages at a faster rate, resulting in a greater number of collisions. This will likely require packets to be retransmitted, resulting in greater latency. Table 3 shows the independent variables of interest, the values for each variable to be investigated during the course of the experiments and the CRTS parameter that controls each variable.

**Table 3. Independent variables for the CORNET experiments**

| Independent Variable | CRTS Parameter | Selected Values |
|---|---|---|
| Bits per Symbol | tx_modulation<br>rx_modulation | 2<br>4<br>8<br>16<br>32 |
| Modulation Scheme | tx_modulation<br>rx_modulation | PSK<br>ASK<br>QAM |
| Data Rate | tx_rate<br>rx_rate | 1 Mbps<br>2 Mbps<br>5.5 Mbps<br>10 Mbps |

Tables 20 through 22 in Appendix A give the variables whose values will remain static throughout the experiments and their associated values. Many of these parameter values are taken from the default CORNET node configuration.

**Important Assumptions and Limiting Factors.**

There are a number of important assumptions made in establishing the experimental design, including:

1. To decrease unintentional interference, the same two adjacent CORNET nodes are used throughout the experiments. While this may not entirely eliminate interference, it is assumed this design choice reduces the strength and duration of any electromagnetic interference with respect to the transmission power and length of exchanged messages such that any residual noise the transmissions do encounter will not appreciably increase latency.

2. CORNET only supports a limited number of modulation schemes by default. While other, more advanced schemes may offer slightly better performance, it is assumed the latency bound determined with the modulation schemes available by default will not differ significantly from that found using a more advanced scheme. Additionally, the schemes investigated here are standard schemes, used in a wide range of applications whereas more advanced schemes may only be applicable to a small number of more advanced scenarios.

3. Messages will be automatically generated by the CORNET nodes. It is assumed they will exhibit sufficient randomness as to approximate normal network traffic such that the latencies observed during the CORNET experiments will not differ significantly from real-world network traffic.

4. Latency measurements will be performed using native CORNET network layer performance metric reporting capabilities and it is assumed these capabilities are of sufficient granularity as to support meaningful latency measurements.

5. Node synchronization will be handled automatically by CORNET via the Network Time Protocol (NTP) and it is assumed the synchronization is of sufficient accuracy as to not invalidate latency measurements.

**Experimental Procedure.**

The CORNET experiments will test the performance of each possible combination of modulation scheme, constellation size and data rate, as shown in Table 4. Before executing any test cases, the necessary CRTS configuration files will be created and pre-staged in the CORNET file system. In order to obtain a sufficient number of samples such that the Central Limit Theorem [54] holds, each test case will run for 200 seconds, in accordance with Table 20. The latencies of all packets sent from one node

and received at the other node during each test case run will be averaged together to determine the latency figure for that case. Additionally, to obtain a single per-packet latency value against which the network's performance with a cognitive engine may be compared, the latency values determined for each test case will be averaged together. The variance for each test case as well as for the latency figure determined by averaging all cases together will be evaluated to determine if the latency values so determined are valid. If not, multiple latency values, representative of different network and node configurations, will be needed and the cognitive network's performance with each of these configurations will have to be compared against the associated latency value individually.

The CORNET nodes used in the experiments must be carefully chosen so as to not introduce additional, unnecessary sources of latency. The selection of the two communicating nodes must ensure they are adjacent to minimize the amount of latency caused by propagation. (If necessary, this could be removed from the data by calculating how long message propagation takes and subtracting this value from all observed data points.) Further, the same two nodes should be used for each test case. Conversely, the selection of the master node, which coordinates each test, is not subject to any constraints. As the master node does not directly take part in the test (i.e., does not create or exchange any messages whose latency will be measured), its location in the network with respect to the other nodes is of no consequence to the latencies observed.

To run the tests, the CRTS software is invoked on the master node and supplied with a master configuration file specifying which scenarios to run. The node then initializes the two communicating nodes with the operating parameters specified in the supplied configuration file for the first scenario. The two communicating nodes then exchange messages and record their timestamps for 200 seconds. At the con-

**Table 4. Node configurations to be tested in CORNET experiments**

| Test Number | Data Rate (Mbps) | Modulation Scheme | Constellation Size |
|---|---|---|---|
| 1 | 1 | ASK-PAM | 2 |
| 2 | 1 | ASK-PAM | 4 |
| 3 | 1 | ASK-PAM | 8 |
| 4 | 1 | ASK-PAM | 16 |
| 5 | 1 | ASK-PAM | 32 |
| 6 | 2 | ASK-PAM | 2 |
| 7 | 2 | ASK-PAM | 4 |
| 8 | 2 | ASK-PAM | 8 |
| 9 | 2 | ASK-PAM | 16 |
| 10 | 2 | ASK-PAM | 32 |
| 11 | 5.5 | ASK-PAM | 2 |
| 12 | 5.5 | ASK-PAM | 4 |
| 13 | 5.5 | ASK-PAM | 8 |
| 14 | 5.5 | ASK-PAM | 16 |
| 15 | 5.5 | ASK-PAM | 32 |
| 16 | 10 | ASK-PAM | 2 |
| 17 | 10 | ASK-PAM | 4 |
| 18 | 10 | ASK-PAM | 8 |
| 19 | 10 | ASK-PAM | 16 |
| 20 | 10 | ASK-PAM | 32 |
| 21 | 1 | PSK | 2 |
| 22 | 1 | PSK | 4 |
| 23 | 1 | PSK | 8 |
| 24 | 1 | PSK | 16 |
| 25 | 1 | PSK | 32 |
| 26 | 2 | PSK | 2 |
| 27 | 2 | PSK | 4 |
| 28 | 2 | PSK | 8 |
| 29 | 2 | PSK | 16 |
| 30 | 2 | PSK | 32 |
| 31 | 5.5 | PSK | 2 |
| 32 | 5.5 | PSK | 4 |
| 33 | 5.5 | PSK | 8 |
| 34 | 5.5 | PSK | 16 |
| 35 | 5.5 | PSK | 32 |
| 36 | 10 | PSK | 2 |
| 37 | 10 | PSK | 4 |
| 38 | 10 | PSK | 8 |
| 39 | 10 | PSK | 16 |
| 40 | 10 | PSK | 32 |
| 41 | 1 | QAM | 4 |
| 42 | 1 | QAM | 8 |
| 43 | 1 | QAM | 16 |
| 44 | 1 | QAM | 32 |
| 45 | 2 | QAM | 4 |
| 46 | 2 | QAM | 8 |
| 47 | 2 | QAM | 16 |
| 48 | 2 | QAM | 32 |
| 49 | 5.5 | QAM | 4 |
| 50 | 5.5 | QAM | 8 |
| 51 | 5.5 | QAM | 16 |
| 52 | 5.5 | QAM | 32 |
| 53 | 10 | QAM | 4 |
| 54 | 10 | QAM | 8 |
| 55 | 10 | QAM | 16 |
| 56 | 10 | QAM | 32 |

clusion of each test, results are automatically written into log files and converted to Octave scripts for MATLAB ingest. Once the first test concludes, the master node reinitializes the communicating nodes with the operating parameters for the second scenario. The nodes then again exchange messages and record results for 200 seconds, writing the data to a log file when the test concludes. This process repeats for all scenarios specified in the master configuration file.

**Data Analysis.**

**Calculating Latencies by Comparing Timestamp Differences Between Transmitter and Receiver.**

CORNET automatically provides a vector of message identification numbers and the timestamps at which the transmitter sends those messages and analogous vectors for the receiver which contain the message identification numbers and the timestamps at which those messages were received. Thus, to measure the latency of a given received message, the timestamp at which it was sent from the transmitter will be subtracted from the timestamp at which it was received at the receiver to obtain a latency measurement for that message. Care must be taken to ensure the timestamps extracted from the two vectors for use in this calculation are for the same message. It is expected that not all messages sent will be received so simply subtracting the two vectors will likely over-estimate many messages' latency by comparing unrelated timestamps. Instead, the indices at which the same message identifier is found in the two message identification number vectors will be used to index into the timestamp vectors and extract the timestamps at which each message was sent and received. Repeating this calculation for all messages exchanged during the course of the experiment and averaging the results will yield a latency figure for the given test case.

Table 5 shows an example of these calculations. The data shown are sample

numbers, not taken from any experimental runs. As messages 4, 6 and 7 were not successfully received, no latency measure for them may be calculated and they are not used in calculating the overall average latency value. For each of the remaining messages, the timestamps at which the message was sent and received are subtracted to yield the latency measure for that message. For example, the transmitter sent message 1 at .01 seconds and the receiver received it at .018 seconds, yielding a latency measure of .008 seconds.

As stated previously, care must be taken to ensure the correct timestamps are extracted and compared. The vectors as given by CORNET do not have a blank entry in the timestamp vector for missed messages; the blanks shown in Table 5 are simply to keep the table aligned. Without using the indices at which the message identifiers are found in the message number vector to index into the timestamp vector, the wrong timestamps may be extracted and compared, yielding erroneously high results. For example, the timestamp for message 5 is found at index 4 (using 0-based indexing) in the transmitter's timestamp vector but it is at index 3 in the receiver's vector. Therefore, only by determining the index at which the message identifier appears in each node's identification number vector and using the timestamp at the same index in the nodes' timestamp vector may the appropriate timestamps be compared.

**Calculating Latencies by Comparing Timestamps at Receiver for Messages with Consecutive Identification Numbers.**

To compare different methods for calculating latency from the CORNET results, the data will also be analyzed a second way. In this approach, the message identification numbers and timestamp vectors for the transmitter will neglected. Instead, the timestamps for packets received with consecutive message numbers will be extracted from the receiver's vector and subtracted to yield a latency measure. That is, if two

**Table 5. Example calculation using the message identification number and timestamp vectors provided by CORNET for the transmitter and receiver**

| Transmitter | | Receiver | | | |
|---|---|---|---|---|---|
| Message Number | Timestamp | Message Number | Timestamp | Message reception status | Difference (s) |
| 1 | .01 | 1 | .018 | Message received Timestamps may be compared | .008 |
| 2 | .015 | 2 | .024 | Message received Timestamps may be compared | .009 |
| 3 | .021 | 3 | .03 | Message received Timestamps may be compared | .009 |
| 4 | .24 | – | – | Message not successfully received Timestamps may not be compared | – |
| 5 | .028 | 5 | .035 | Message received Timestamps may be compared | .007 |
| 6 | .034 | – | – | Message not successfully received Timestamps may not be compared | – |
| 7 | .039 | – | – | Message not successfully received Timestamps may not be compared | – |
| 8 | .043 | 8 | .052 | Message received Timestamps may be compared | .009 |
| 9 | .049 | 9 | .058 | Message received Timestamps may be compared | .009 |
| 10 | .054 | 10 | .061 | Message received Timestamps may be compared | .007 |
| | | | | Average: | ∼.00829 |

messages appear in the message number vector and the difference between their identifiers is 1, their associated timestamps will be extracted from the timestamp vector and subtracted to yield a latency measure. Conversely, if the message numbers differ by more than one, their associated timestamps will not be compared. This method will not directly measure the latency of any given message but instead will use the time lapse between successfully received consecutive messages as a surrogate. This calculation will be repeated for all packets received with consecutive message identification numbers and the results averaged together to obtain the latency figure for that test case.

Table 6 shows an example of this data analysis method. It should be noted that, as with the previous example, the data shown are sample numbers, not actual results. Due to the receiver missing several messages, only 5 latency measures may be calculated from the data. These measures come from message pairs consisting of messages 1 and 2, 4 and 5, 5 and 6, 9 and 10, and 14 and 15. As shown in the table, the receiver missed at least one message in all other possible pairs of consecutive messages. To determine the latency measures for each of these message pairs, the timestamp at which the messages with the smaller identifier was received is subtracted from the

timestamp at which the message with the larger identifier was received. The resulting latency measures are then averaged together to yield a latency figure for the test case.

**Calculating Latencies by Comparing Timestamps at Receiver for Consecutively-Received Messages.**

The CORNET data sets will also be analyzed using a third analysis method in an attempt to include some of the data neglected by the previous data analysis method. Here again, only the message identification number and timestamp vectors for the receiver will be used. However, rather than requiring message identification numbers be consecutive (i.e., have a difference of 1) in order to use their associated timestamps to obtain a latency measure, all pairs of messages received in order at the receiver will be used to calculate latency measures. Each timestamp will be subtracted from the one immediately preceding it to yield a time difference. Then, the two message identification numbers associated with the two timestamps will be subtracted from each other and the time delta divided by this number. The result will be used as the latency measure and all measures thus determined will be averaged together for that test case.

As with the preceding two analysis methods, Table 7 shows an example of this technique. In comparison to Table 6, from Table 7 it may immediately be seen that anytime there is a gap in the identification numbers of the messages received, the first message received after the gap still has a latency measure associated with it. For example, in Table 6, message 9 does not have a latency measure associated with it because it was the first message received after the gap in which the receiver missed messages 6, 7 and 8. In Table 7, however, message 9 still has a latency measure associated with it. The difference between the time at which the receiver received message 9 and the preceding message (6) is simply divided by one more than the

**Table 6. Example calculation using timestamps for messages received with consecutive message identification**

| Receiver | | | |
|---|---|---|---|
| Message Number | Timestamp | Message reception status | Difference (s) |
| 1 | .02 | First message received<br>No previous message to which to compare | – |
| 2 | .025 | Message number consecutive<br>Timestamps may be compared | .005 |
| – | – | Message not received<br>Timestamps may not be compared | – |
| 4 | .036 | Preceding message missed<br>No preceding message to which to compare | – |
| 5 | .04 | Message number consecutive<br>Timestamps may be compared | .004 |
| 6 | .046 | Message number consecutive<br>Timestamps may be compared | .006 |
| – | – | Message not received<br>Timestamps may not be compared | – |
| – | – | Message not received<br>Timestamps may not be compared | – |
| – | – | Message not received<br>Timestamps may not be compared | – |
| 9 | .063 | Preceding message missed<br>No preceding message to which to compare | – |
| 10 | .068 | Message number consecutive<br>Timestamps may be compared | .005 |
| – | – | Message not successfully received<br>Timestamps may not be compared | – |
| – | – | Message not successfully received<br>Timestamps may not be compared | – |
| – | – | Message not successfully received<br>Timestamps may not be compared | – |
| 14 | .092 | Preceding message missed<br>No preceding message to which to compare | – |
| 15 | .098 | Message number consecutive<br>Timestamps may be compared | .006 |
| | | Average: | .0052 |

number of messages missed. For message 9, the difference between the time at which the receiver received it and the preceding message is .017 seconds (.063 - .046 seconds). Further, the receiver missed 2 messages in between messages 6 and 9. The difference between the times at which the receiver recieved them (.017 seconds) is therefore divided by 3. This calculation is repeated for all pairs of consecutively received messages. The resulting latency measures are then averaged together to obtain the latency measure for the test case.

## 4.2 OMNeT++ Engine Performance Tests

The main set of experiments that will be conducted as part of this research involve assessing the performance of the cognitive engine developed in the preceding chapter with respect to network latency. The effect the engine has on the network's latency as the noise floor increases is of interest. In particular, the experiments that will be

**Table 7. Example calculation using timestamps for messages received regardless of the difference in the associated message identification numbers**

| Receiver | | | |
|---|---|---|---|
| Message Number | Timestamp | Number by which timestamp difference is to be divided (Number of missed messages plus 1) | Difference (s) |
| 1 | .02 | First message received<br>No previous message to which to compare | – |
| 2 | .025 | 1 | .005 |
| – | – | – | – |
| 4 | .036 | 2 | .0055 |
| 5 | .04 | 1 | .004 |
| 6 | .046 | 1 | .006 |
| – | – | Message not received<br>Timestamps may not be compared | – |
| – | – | Message not received<br>Timestamps may not be compared | – |
| – | – | Message not received<br>Timestamps may not be compared | – |
| 9 | .063 | 3 | ∼.00567 |
| 10 | .068 | 1 | .005 |
| – | – | Message not successfully received<br>Timestamps may not be compared | – |
| – | – | Message not successfully received<br>Timestamps may not be compared | – |
| – | – | Message not successfully received<br>Timestamps may not be compared | – |
| 14 | .092 | 4 | .006 |
| 15 | .098 | 1 | .006 |
| | | Average: | ∼.00540 |

conducted seek to determine if the engine can tune the network in such a manner as to achieve lower latencies as the noise floor increases compared to a network without a cognitive engine.

### Independent and Dependent Variables.

For these experiments, the independent variables of interest are latency weight, noise floor and number of generations. The latency weight indicates the priority given to the latency component in the fitness function. As it increases, solutions that emphasize lower Bit Error Rates (BERs) to improve latency performance are assigned higher fitness values than solutions that trade some latency (and throughput) performance to reduce power consumption. Therefore, as the latency weight changes, it is expected the solutions returned by the engine will change and the latencies realized in the network will decrease. As a result, it is expected the cognitive network will have increasingly improved latency performance as compared to its non-cognitive counterpart as the latency weight increases.

As the number of generations the engine uses to develop solutions increases, it

potentially allows the engine to search previously unexplored regions of the search space (through global search) or find better solutions in the vicinity of previously found "good" solutions (through local search). It is expected, then, that the fitness of solutions returned by engine as the number of generations it uses to evolve solutions increases will, in general, improve. Thus, as the number of generations the engine uses grows, it is expected the network's latency performance will continually increase as compared to the non-cognitive, baseline network.

Additionally, it is also expected that as the number of generations increases, the engine's runtime will also increase. The rate at which this increase occurs is of interest. Here, the engine has arbitrarily been set to run every 5 seconds. However, depending on how fast conditions in the radio medium are expected to change, this parameter could be set to a wide range of values. Therefore, how many generations may be developed, and thus the portion of the search space that may be explored in a given amount of time, is of interest.

Table 8 gives the minimum and maximum values as well as the step size for the independent variables of interest.

**Table 8. Independent variables, including minimum and maximum values and step size, for the OMNeT++ experiments**

| Independent Variable | Minimum Value | Maximum Value | Step Size |
|---|---|---|---|
| Latency Weight | .5 | .9 | .1 |
| Number of Generations | 200 | 1000 | 200 |
| Noise Floor (dBm) | -90 | 0 | 10 |

### Important Assumptions and Limiting Factors.

As with the CORNET experiments, there are a number of assumptions and other factors that must be taken into account, including

1. Newtork traffic generation will be handled by the OMNeT++ framework. How-

ever, the pseudo-random number generator (PNG) the software uses in generating the traffic will be initialized to a different value for each experimental run to ensure the same exact sequence of packets is not produced for each test case. Beyond this, it is assumed the traffic characteristics (e.g., inter-packet delays, etc.) are sufficiently random as to approximate real-world network traffic.

2. Rather than having a node in the network to produce interference, the background noise level is incremented throughout the test runs to simulate the effect of a broadband jammer. It is assumed varying the background noise level in this manner has a similar effect on the network nodes and their ability to communicate as a broadband jammer periodically increasing its output power would have.

3. As with CORNET, OMNeT++ only supports a limited number of modulation schemes by default. Here, only PSK and QAM are considered. While other, more advanced schemes may offer slightly better performance, it is assumed the trends noted in the network's latency performance with and without the engine will not differ significantly from that found using a more advanced scheme.

4. Latency measurements will be taken as the end-to-end delay metric reported by OMNeT++'s "UDPSink" module. It is assumed this metric, which reports the latency for network-layer packets exchanged between nodes, is of sufficient granularity as to support meaningful latency measurements.

**Experimental Procedure.**

To investigate each of the independent variables outlined previously and their effect on the network's latency performance, the network is first run without the cognitive engine to collect baseline latency statistics. The engine is then added to

the network and the simulation run ten times, once for each value of latency weight and number of generations used to develop solutions, as outlined in Table 8. Each simulation is run for 2,000 seconds, with the noise floor starting at -90 dBm and incrementing by 10 dBm every 200 seconds. As the engine runs every 5 seconds, this ensures it runs 40 times for each combination of latency weight or number of generations and noise floor to provide adequate data.

In order to evaluate how the number of generations used to develop solutions affects the engine's runtime, additional tests wherein the engine is removed from the network and run in isolation are also performed. A simple test program that invokes the engine and times its execution is written. The program executes the engine 40 times for each number of generations value given in Table 8 and records the engine's runtime for each execution.

**Data Analysis.**

In investigating how the number of generations affects network performance, the latency data resulting from the five number of generations runs must first be split into ten sets based on the noise floor. For each noise floor, then, there are five data sets, one for each setting of the number of generations parameter, giving network latencies against simulation time. A simple t-test is performed to compare the average latency without the CE in the network to that achieved with the engine in the network for each investigated value for the number of generations. A p-value less than or equal to .05 from this test indicates the engine has an appreciable effect on the network's performance.

The effect latency weight has on network performance is evaluated in a similar manner. The data resulting from the five simulations across which latency weight was altered is first split into ten data sets based on noise floor. t-tests comparing the

average latency without an engine in the network to the network's performance with each of the investigated latency weights are performed. As before, a p-value less than or equal to .05 from these tests indicate the engine with the associated latency weight has an appreciable effect on the network's performance.

By comparison, the data analysis performed on the engine's runtime data is straightforward. As it is expected the runtime will increase in proportion to the number of generations, a linear model is fit to the runtime vice number of generations data. Then, the p-value associated with the number of generations variable is assessed to determine if it statistically significant within the model (i.e., if its value is less than or equal to .05). If it is statistically significant, this indicates the number of generations the engine uses to develop solutions has a direct and substantial impact on its runtime. Additionally, the resulting linear model may be used to determine, given a desired runtime, approximately how many generations may be developed by the engine in that time.

## 4.3 MATLAB Cognitive Engine Performance Tests

Due to issues with the OMNeT++ tests detailed in Chapter V, the CE is reimplemented and tested in MATLAB. All of the engine's functionality, save for the manner in which the BER is calculated, remains unchanged from the development given in Chapter III. (The code for the engine's MATLAB reimplementation is given in Appendix C.) In the reimplementation, rather than evaluating functions to estimate BER, the engine simulates the transmission of bits and directly calculates the number of bits received in error. To do this, the engine uses built-in MATLAB functionality. In particular, MATLAB randomly generates a string of bits to "transmit". The bits are then passed through a modulator, Additive White Gaussian Noise (AWGN) channel and a demodulator, provided by MATLAB as system objects. Fur-

ther, as MATLAB provides built-in functionality for performing FEC encoding and decoding unlike the network as built in OMNeT++, in some of the tests, the bits are also FEC encoded before modulation and FEC decoded after demodulation. In this way, the engine's performance with and without FEC encoding may be investigated. The number of bit errors that results from the bit transmission divided by the total number of transmitted bits is used as the BER.

Figure 17 shows a plot of the latency fitness values returned by the latency fitness function against BER. From the plot, BERs close to 0 result in high latency fitness but as the BER increases by orders of magnitude to $\sim 10^{-4}$, the latency fitness quickly drops to 0. Therefore, in order to detect BERs that result in reasonably high (but not perfect) latency fitness, the number of bits for which transmission is simulated must be on the order of one million bits. As the inputs to the (possible) FEC encoder, modulator, AWGN channel, demodulator and (possible) FEC decoder change for each generation the engine produces, the simulated bit transmissions must be performed each time the engine forms a new solution in order to assign a fitness value to it. Unfortunately, the time required to perform the simulated bit transmission so often causes the MATLAB runtime to greatly exceed that of the OMNeT++ simulations. Additionally, the OMNeT++ network yielded actual latency information whereas the engine reimplementation in MATLAB only yields latency and total fitness information. As a result, the experiments performed and data gathered from the MATLAB engine reimplementation have a slightly different focus than those planned for the OMNeT++ simulations.

**Independent and Dependent Variables.**

For the MATLAB engine reimplementation, there are several major independent variables under test. Two of these variables, FEC encoding and number of genera-

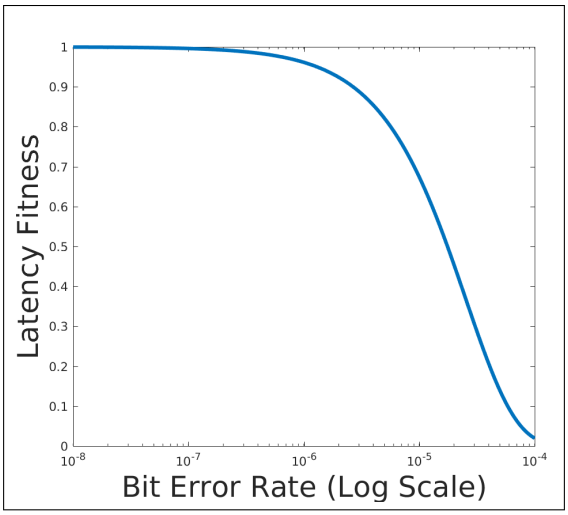


**Figure 17. Plot showing latency fitness values against bit error rate**

tions, will be evaluated through running multiple stochastic experiments while the effect of the remaining variables, modulation scheme and order, power and bandwidth, will be evaluated through comparing the resulting fitness values as these parameters change from their optimum values.

The first variable to be assessed through experimentation is FEC encoding. In particular, rather than evaluating multiple schemes to determine the one with which the engine performs best, here the item of interest is whether or not the engine performs better with or without FEC encoding. It is expected FEC encoding will cause an increase in the engine's performance, evidenced by the engine returning solutions with lower BERs and higher latency fitness than those returned for the same non-FEC encoded case. However, it is also expected FEC encoding will increase the runtime as compared to the non-FEC encoded case. Further, this effect is expected to be more pronounced as the noise floor becomes closer to the transmit power (i.e., as the Signal-to-Noise Ratio (SNR) drops). If this trend is indeed exhibited by the data, future work will need to be conducted to determine at what point FEC encoding offers enough of a performance improvement that the penalty to runtime is worth the

improved fitness values.

Similar to the OMNeT++ simulations, another independent variable of interest here is the number of generations used by the engine and its effect on the quality of the solutions the engine returns. As discussed above, due to the manner in which the MATLAB engine reimplementation estimates BER, the number of generations is expected to have a significant impact on the engine's runtime. If this is indeed the case, as with the previous experiments, determining the point at which the increase in runtime does not yield enough improvement in the fitness of returned solutions to warrant using additional generations in finding them becomes an important part of future work. Additionally, the runtime of the MATLAB engine implementation may be compared to the C++ engine implementation included in the OMNeT++ network to comment on the importance of careful engine design due to the effect implementation decisions (e.g., coding language, data structures, variable declarations and calculation approaches, etc.) may have on the runtime of an actual engine included in a real-world Unmanned Aerial Vehicle (UAV) swarm.

The next four independent variables of interest are the four tuning knobs made available to the engine: modulation scheme, modulation order, power and bandwidth. Due to the small size of the search space used in the engine developed here, it may be exhaustively searched to find the optimum solution. (Recommendations for increasing the size of the search space in order to take greater advantage of the power of the genetic algorithm on which the engine is based are included in Chapter VI.) As each variable deviates from its optimum setting, the fitness of the resulting solution will decrease. However, the rate at which this deviation from optimum fitness occurs is of interest. To explore this, the resulting fitness values as each parameter varies through its entire range of settings while the other values remain constant at their optimum settings will be considered.

The last independent variable of interest is the noise floor. The effect of the independent variables previously outlined will be assessed at three different noise floors. The independent variables for the MATLAB engine implementation experiments are summarized in Table 9.

**Table 9. Independent variables, including minimum and maximum values and step size, for the MATLAB engine reimplementation experiments**

| Independent Variable | Minimum Value | Maximum Value | Step Size |
|---|---|---|---|
| FEC Encoding | NA | NA | |
| Number of Generations | 200 | 1000 | 400 |
| Noise Floor (dBm) | -50 | 10 | 30 |
| Modulation Scheme | NA | NA | NA |
| Bits Per Symbol | 1 | 8 | 1 |
| Power (dB) | 4 | 35 | 1 |
| Bandwidth (MHz) | 1 | 32 | 1 |

### Important Assumptions and Limiting Factors.

There is one major assumption associated with the engine reimplementation under test here:

1. The MATLAB calculations performed to estimate BER are the result of mathematical manipulations of a randomly-generated bit string, not the result of any actual transmissions. However, the MATLAB functionality used in these calculations is based on sound signal processing theory and has been shown to approximate real-world performance with enough fidelity that it is assumed their use here does not invalidate the data collected nor the conclusions derived from that data.

**Experimental Procedure.**

To determine the effect FEC encoding, the number of generations and the noise floor has on engine performance, the engine is run with each possible combination of the independent variables of interest. Table 10 outlines each of these combinations. Each case is run 30 times. The resulting data is then processed in multiple ways to illuminate the effect each of the variables has on the quality of solutions and engine runtime, as discussed below.

**Table 10. Test cases for determining effect of FEC encoding, number of generations and noise floor on engine performance**

| FEC Encoded? | Number of Generations | Noise Floor (dBm) |
|---|---|---|
| No | 200 | -50 |
| Yes | 200 | -50 |
| No | 600 | -50 |
| Yes | 600 | -50 |
| No | 1000 | -50 |
| Yes | 1000 | -50 |
| No | 200 | -30 |
| Yes | 200 | -30 |
| No | 600 | -30 |
| Yes | 600 | -30 |
| No | 1000 | -30 |
| Yes | 1000 | -30 |
| No | 200 | 10 |
| Yes | 200 | 10 |
| No | 600 | 10 |
| Yes | 600 | 10 |
| No | 1000 | 10 |
| Yes | 1000 | 10 |

The engine's ability to find good solutions, not just the actual fitness of the re-

turned solutions, is assessed by comparing the optimal setting for each parameter to how often the engine returns that value for the parameter. To do this, an exhaustive search is performed at each noise floor level with and without FEC encoding to find the optimal solution for that noise floor and FEC encoding setting combination. (As this is an exhaustive search, a setting for the number of generations is not applicable here; all possible solutions will be considered.) Once the optimum solution for each combination of noise floor and FEC encoding setting is known, the solutions returned by the 270 runs previously outlined are evaluated to determine how often the optimum value for each parameter is returned. Additionally, the effect each parameter has on the solution's fitness is investigated by varying each in turn through their entire range of possible values while holding the others constant at their optimum value and plotting the resulting fitness values.

**Data Analysis.**

To assess the effect FEC encoding has on the quality of solutions and the engine's runtime, a single data table containing all fitness and runtime vice number of generations data is compiled. Then, a cateogrical variable that encodes whether or not a given test case included FEC is added to the table. Next, a model is fit to the data, taking into account both the number of generations and the categorical variable encoding whether or not FEC was used. The p-values associated with the coefficients for these two variables in the resulting model are assessed to determine whether or not the variables are significant. In particular, a p-value less than or equal to .05 for the categorical variable coefficient indicates FEC encoding has a statistically significant impact on fitness and/or runtime (depending on the model for which the p-value was significant). It is not expected FEC encoding causes a significant difference in the rate at which solution fitness grows but that it does have an appreciable impact on

runtime growth.

To determine the effect the number of generations has on the quality of returned solutions and the engine runtime, the solution fitness and engine runtime vice number of generations data will be plotted for each noise floor both with and without FEC, resulting in two plots. Linear models relating the fitness and runtime to the number of generations will then be fit to the data. How the number of generations affects solution fitness and engine runtime will be assessed by evaluating the p-values associated with the linear term coefficient in the models. A significant p-value (less than or equal to .05) indicates the number of generations has a statistically significant impact on solution fitness and/or runtime (depending on the model for which the p-values are significant). It is expected solution fitness will increase with the number of generations but that engine runtime will also increase.

The number of times the engine returns each of the possible values for each parameter at each noise floor with and without FEC encoding is plotted on a bar graph, resulting in eight plots total. These are then compared to the optimum solution as determined by exhaustive search to determine the percentage of time the engine returns the optimum setting for each parameter, given the noise floor and FEC encoding setting. Graphs of how the fitness values vary as each parameter takes on each of its possible values while all other parameters are held at their optimum value are then produced to gain insight into how great an effect the engine not returning the optimum value for a given parameter has on the fitness of the returned solution.

# V. Results and Analysis

## 5.1 CORNET Latency Lower-Bound Experimental Results

Before discussing the results obtained from the CORNET experiments, it is useful to state what the data were expected to show. It was expected the calculated latencies would be randomly scattered about some average value according to an approximately normal distribution. An example of what the plot of latencies vice message number was expected to look like is shown in Figure 18. Further, Figure 19 shows the histogram of the data shown in the scatter plot, from which it is easier to see the data follow an approximately normal distribution.

**Calculating Latencies by Comparing Timestamp Differences Between Transmitter and Receiver.**

In comparison with Figure 18, Figures 20 and 21 show the two major data trends that result from processing the CORNET output by comparing the timestamp vectors from the transmitter and receiver. (As there are 56 total tests cases, only two plots are shown here. However, all but one of the remaining plots have a shape similar to Figure 20 while the last plot has a shape similar to Figure 21.) While it is unclear precisely why the data exhibits such unexpected behavior, the most likely explanation, one suggested by the shape of the data itself, is there is unexpected and undesirable queueing occurring within the system that is masking the true latency values.

For example, in explaining the initial linear growth in Figure 20, it is possible the transmitter is preparing packets for transmission, adding a timestamp to them and then placing them in a queue for transmission. If this is indeed how the transmitter operates, it is likely the processing required to prepare a packet for transmission can occur at a faster rate than the actual transmission. As a result, the packet queue

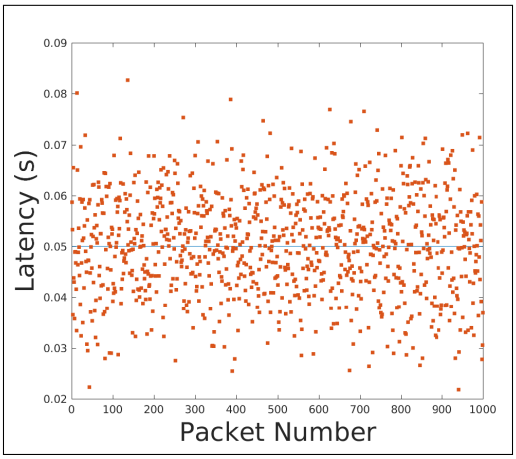


**Figure 18. Example plot of CORNET results showing latencies versus message number**

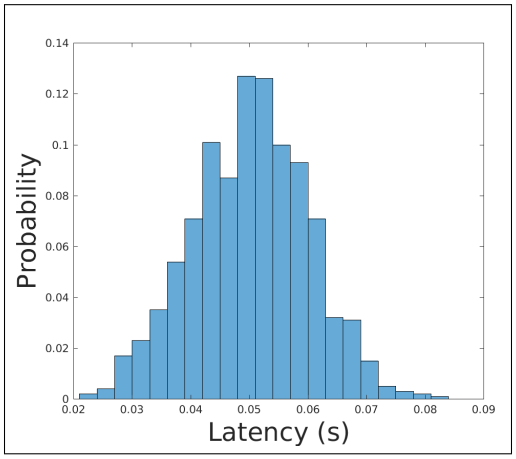


**Figure 19. Example histogram of CORNET results showing probability of latency values in each bin occurring**

awaiting transmission would grow according to the difference between the rate at which packets are prepared for transmission and placed in the queue and the rate at which the packets are removed from the queue and transmitted. This would explain the linear growth in the latencies; each packet in the queue must wait for all packets preceding it to be transmitted before it is transmitted. Thus, as the queue grows, the packets waiting in it must wait for progressively longer periods of time.

If this is the case, it still does not explain why two of the test cases, shown in Table 11, should exhibit the trend noted in Figure 21. The varying growth rate indicates that either the rate at which the transmitter prepared packets for transmission, the transmission rate or both varied widely over the course of the experiment. Why either or both of these rates should have varied at all over the course of the experiment is unclear. Further, this trend only appears in two of the Amplitude Shift Keying (ASK) cases using a constellation size of 32. It is unlikely the modulation scheme, constellation size or data rate provide an adequate explanation for the data. The constellation size affects the amount of information included in a given transmitted symbol, not the rate at which the symbols may be transmitted (dictated by the transmission rate). Further, other test cases using ASK, a constellation size of 32 and/or a data rate of 2 and 10 Mbps exist that do not exhibit the varying growth rate seen in Figure 21.

**Table 11. Two cases in the CORNET results exhibit initial growth that is approximately linear, though over short time intervals the growth rate varies widely**

| Modulation Scheme | Data Rate (Mbps) | Constellation Size |
|---|---|---|
| ASK | 2 | 32 |
| ASK | 10 | 32 |

The shift from linear growth to a constant latency part way through the test run is more difficult to explain. The change suggests the transmitter altered either the rate

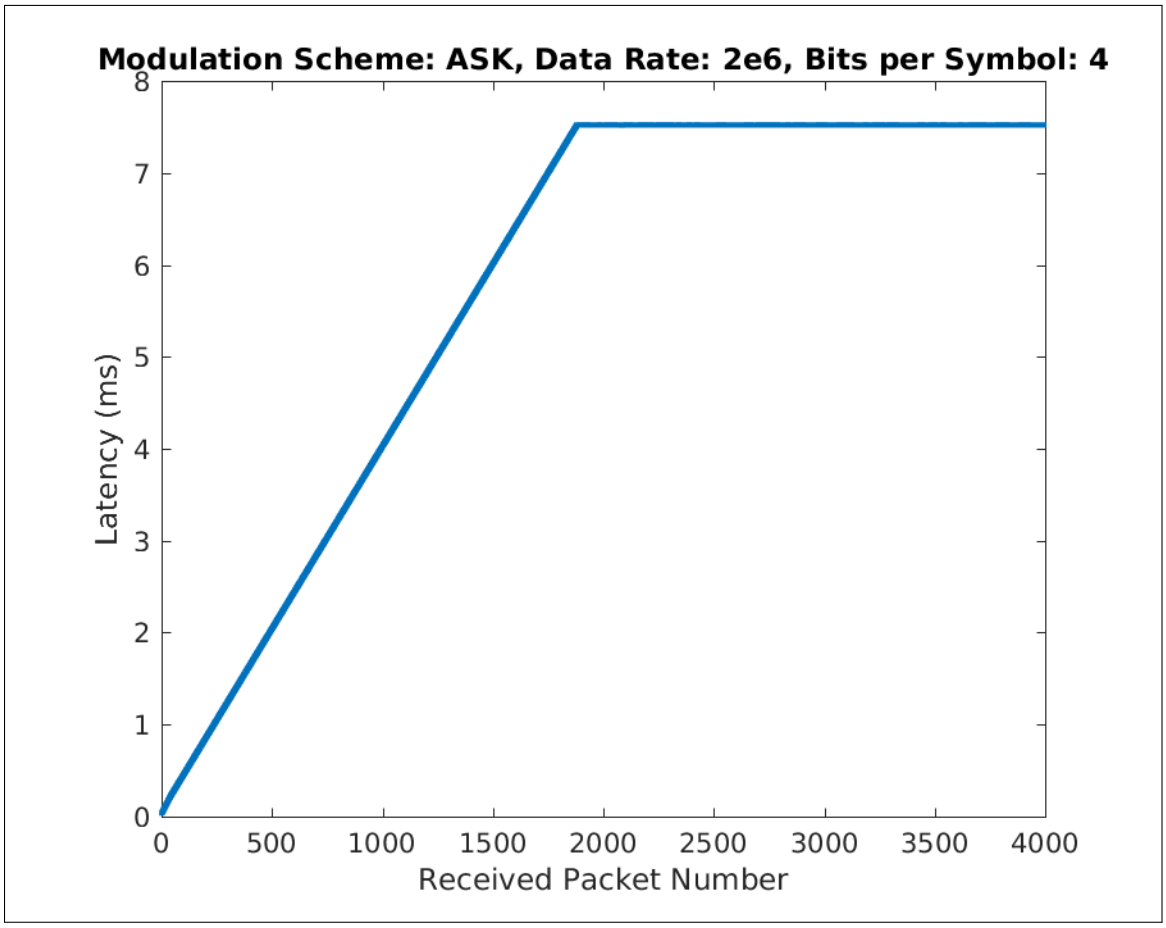


**Figure 20. Results obtained by comparing transmitter and receiver timestamp vectors**

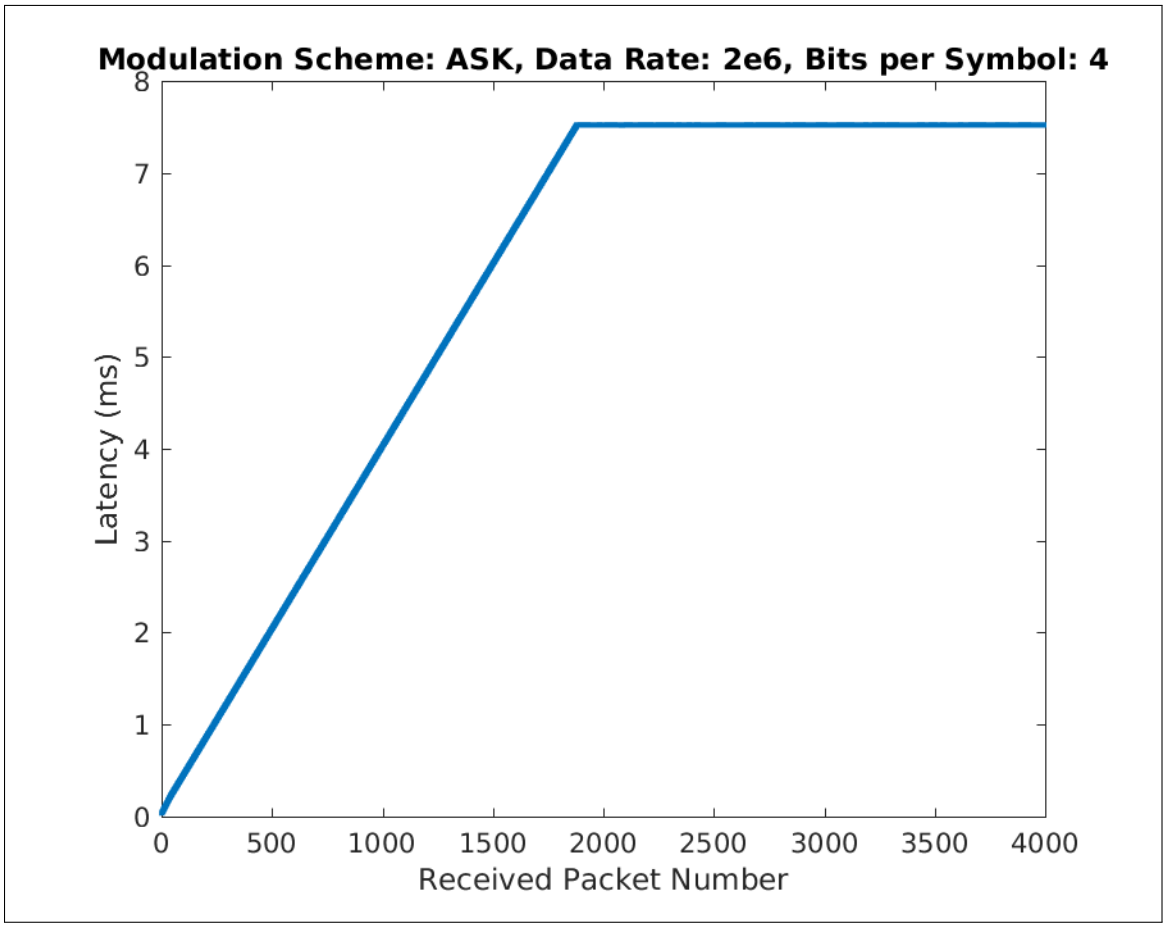


**Figure 21. Results obtained by comparing transmitter and receiver timestamp vectors**

at which it prepared packets for transmission, its transmission rate, or both, to match part way through the test run. It is somewhat unlikely the transmission rate changed; it is set by the user in the scenario configuration file and none of the CORNET documentation indicates the nodes may change any of their operational parameters of their own volition. Further, these experiments did not include a cognitive engine that was trying to optimize the nodes' performance so it does not appear an external entity commanded the transmitter to alter its transmission rate. As for the rate at which the transmitter prepared packets for transmission, if the transmitter did change that rate, it appears to have done so of its own accord. If the transmitter did make the decision to change its operating parameters on its own, it would make measuring latency more difficult.

Analogous queueing within the receiver could also explain the initial linear growth in the latencies. If the receiver has a queue in which it places incoming messages awaiting processing and the rate at which it receives messages is faster than the rate at which it can process them, the amount of time messages wait in the queue to be processed would grow linearly according to the difference in the rate at which they arrive and are processed. This is at least possible; the computation required for processing the received signal to extract the message is considerably more than that required to prepare the message for transmission. As a result, while the transmitter can likely prepare messages faster than it can transmit them, the receiver likely cannot process the incoming signal to recover the communicated information as fast as it arrives. This, however, does not explain the varying growth rate in the two cases in Table 11. Additionally, the same problem arises here as with the transmitter in explaining the shift to a constant latency. If processing with the receiver explains the data trends, it is unclear why the receiver should have altered its message processing rates or why the message arrival rate should have changed.

Other explanations of the data are possible. A combination of the above factors could be the culprit behind the unexpected data. Unanticipated interference in the transmission could be partly to blame. Unknown and unanticipated processing within CORNET (e.g., nodes varying the operational parameters in an unexpected way) could also have skewed the results. Unfortunately, without direct access to the system and/or additional insight into its operation, further explanation of these data trends and use of CORNET for latency measurements is problematic at best.

**Calculating Latencies by Comparing Timestamps at Receiver for Messages with Consecutive Identification Numbers.**

Similar to processing the CORNET results by comparing the timestamp vectors for the transmitter and receiver, generating latency measures by only considering messages received by the receiver with consecutive identification numbers yields unexpected results. These results again follow one of two general trends. As for the previous data analysis technique, only two representative plots are used to demonstrate the trends the data exhibits. Figure 22 shows the first of the two trends. At first glance, the plot looks similar to that in Figure 18, indicating this method of data analysis might yield useful and plausible results. However, the plot does not tell the whole story. By comparing the data included in the plot to the full data set (Figure 24), it becomes apparent about 80 percent of the data is neglected when ignoring messages that are not received with consecutive identification numbers. After an initial period in which the receiver does not miss any messages, it misses at least one messages between every two messages it successfully receives. The mostly likely reason the receiver missed these messages is due to unanticipated interference between the two nodes. However, why the receiver didn't miss any messages initially and then missed at least one message between every two successfully received is more

difficult to explain. One possibility is another transmitter started broadcasting part way through the experiment, though the relatively low power levels used in CORNET and the close proximity of the nodes involved in the experiments make this unlikely. Another possibility is that the transmitter changed its operating parameters, most notably power and gain, part way through the experiment. However, as the transmitter is not able to change operating parameters on its own and the experiment did not utilize any cognitive engine to optimize node performance, this is also unlikely.

The second trend exhibited by the data when processed by only considering messages received with consecutive identification numbers is a banding effect in which small portions of the latencies are clustered around .01 seconds while the remaining data is on the order of $10^{-4}$ to $10^{-6}$. Figure 23 shows a test case that exhibits this behavior. Why the latencies between successive messages should jump back and forth between two relatively constant values for discrete periods of time is not immediately clear. The most likely explanation is that control information is periodically sent over the network and transmitting, receiving, processing and acting on this information requires additional computation (thus higher latencies) beyond normal traffic.

Other explanations of the data are again possible. A combination of the above factors could be the culprit. Unknown and unanticipated processing within CORNET (e.g., nodes varying the operational parameters in an unexpected way) could again have skewed the results. Unfortunately, as with the previous data analysis technique, without direct access to the system and/or additional insight into its operation, further explanation of these data trends and use of CORNET for latency measurements is problematic at best.

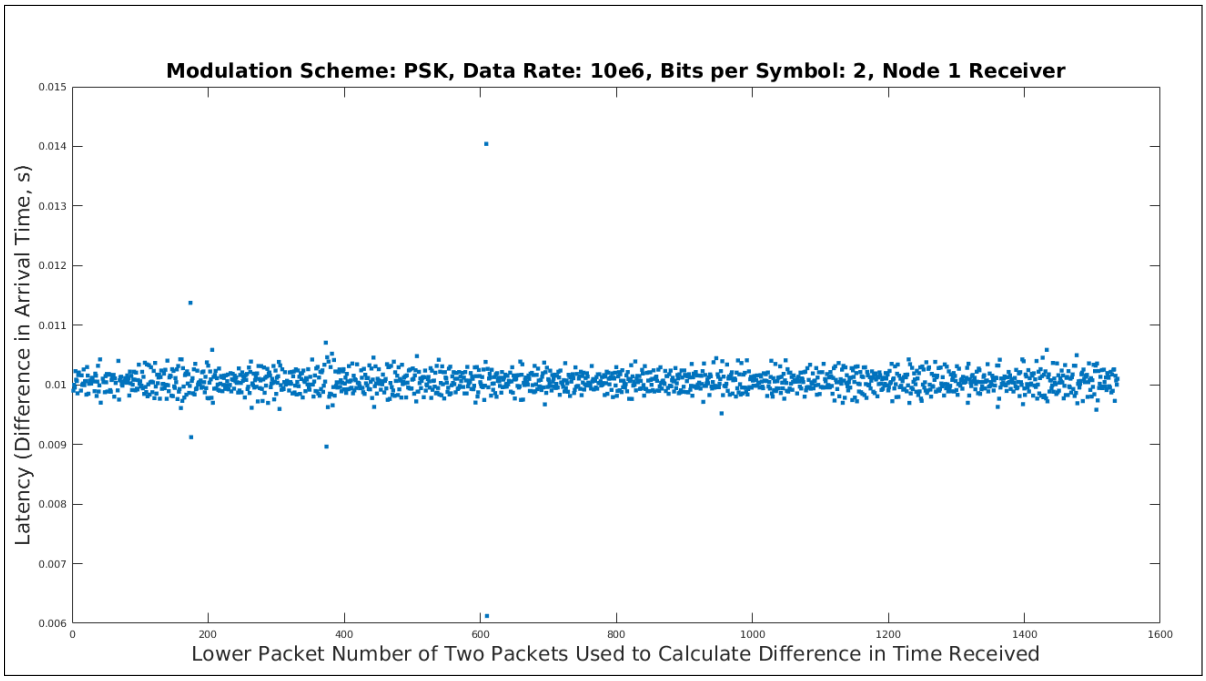


**Figure 22. Results obtained by only using messages received with consecutive numbers**

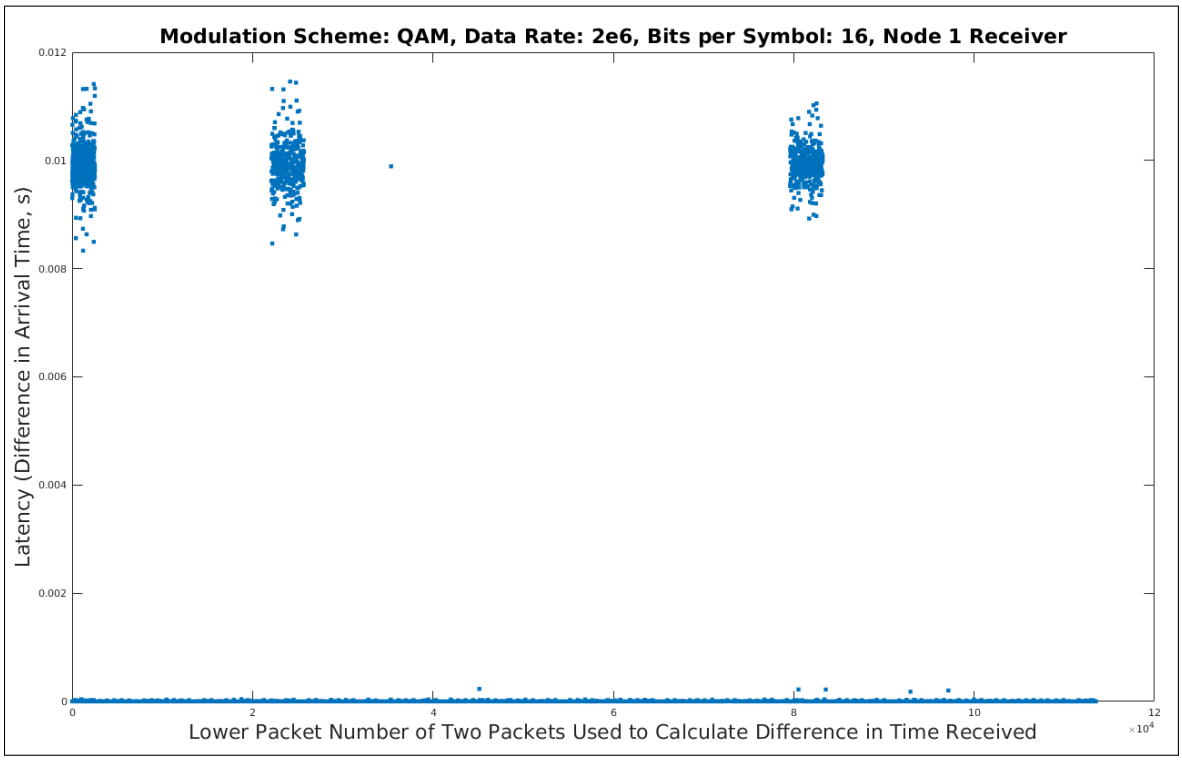


**Figure 23. Results obtained by only using messages received with consecutive numbers**

**Calculating Latencies by Comparing Timestamps at Receiver for Consecutively-Received Messages.**

As with the preceding two data processing techniques, generating latency measures by considering the timestamps of all pairs of messages received consecutively at the receiver yields unexpected results. These results again follow one of two general trends. The first trend, shown in Figure 24, includes the data excluded by the preceding data analysis technique. Importantly, Figure 24 shows the same test case as Figure 22. By comparing these two figures, it is clear how large the portion of the data set excluded by the previous data analysis technique is. This demonstrates why the results from the previous processing technique are questionable; it is difficult to draw conclusions about average values demonstrated by data when the majority of the data is not used in calculating the average values.

The second trend, given in Figure 25 for the same test case as in Figure 23, shows a different banding behavior as seen in the previous processing technique. The data processed by considering all pairs of consecutively received messages still show small sets of latency values clustered around .01 seconds and larger sets of latency values clustered aroud $10^{-4}$ to $10^{-6}$. Here, however, the data also show several distinct bands of latency values clustered around .002 seconds. This indicates there are distinct sets of messages with consecutive identification numbers with latencies around .01, a much larger portion of the messages with consecutive identification numbers with latencies between $10^{-4}$ to $10^{-6}$, but also a considerable number of messages consecutively received, but not with consecutive identification numbers, with latencies around .002 seconds. While the exchange of control information may again explain the messages received with consecutive identification numbers and the distinct difference between the two average values, it does not also explain the bands clustered around .002 seconds. If this data were control information, it would be expected to be clustered

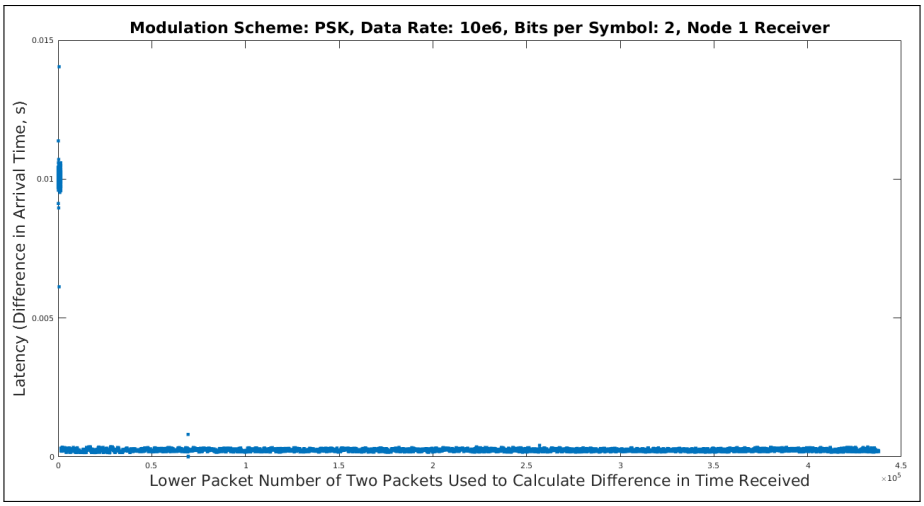


**Figure 24. Results showing the same case as in Figure 22 when the data is processed by averaging the difference in timestamps**

around .01 seconds. Conversely, if the data were normal network traffic taken from some normal distribution with an average value, it would be expected that, after calculating the difference in timestamps and dividing by one more than the number of missed messages, this data would be relatively indistinguishable from the rest of the normal traffic data. As the bands are still distinguishable, this indicates there is a difference between this data and the normal traffic data. In particular, as the number of missed messages increases, the latency does not grow by the same amount for each additional message missed. Instead, the latency grows by a multiplier that is either larger or smaller than one. (Determining if this multiplier is greater than or less than one would require extracting these bands from the data and then determining the timestamps and number of missed messages associated with each data point. As the data is already questionable at this point in the processing, this step is not done here.) However, why these data should be different and what they indicate in terms of how the nodes behaved during the experiment remains unclear at this point.

As with the preceding two processing techniques, other explanations of the data are again possible. A combination of the above factors could explain the data trends

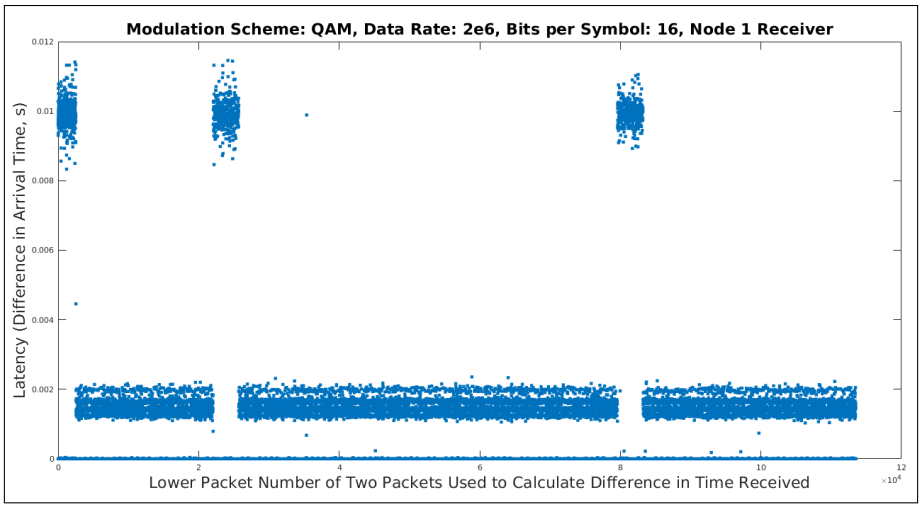


**Figure 25. Results showing the same case as in Figure 23 when the data is processed by averaging the difference in timestamps**

noted. Unknown and unanticipated behavior within CORNET (e.g., nodes varying their operational parameters in an unexpected way) could have led to unexpected results. Unfortunately, without direct access to the system and/or additional insight into its operation, further explanation of these data trends and use of CORNET for latency measurements is problematic at best.

## 5.2 OMNeT++ Engine Experiments

The OMNeT++ experiments aimed to assess the cognitive engine's performance in terms of runtime and its effect on the network's latency. Figure 26 shows the engine's runtime growth with respect to the number of generations it used to evolve solutions. Table 12 shows the model's linear term coefficient and its associated p-value. From the table, because the p-value is less than .05, it may be concluded the coefficient is statistically significant and that the number of generations positively affects the engine's runtime, as expected.

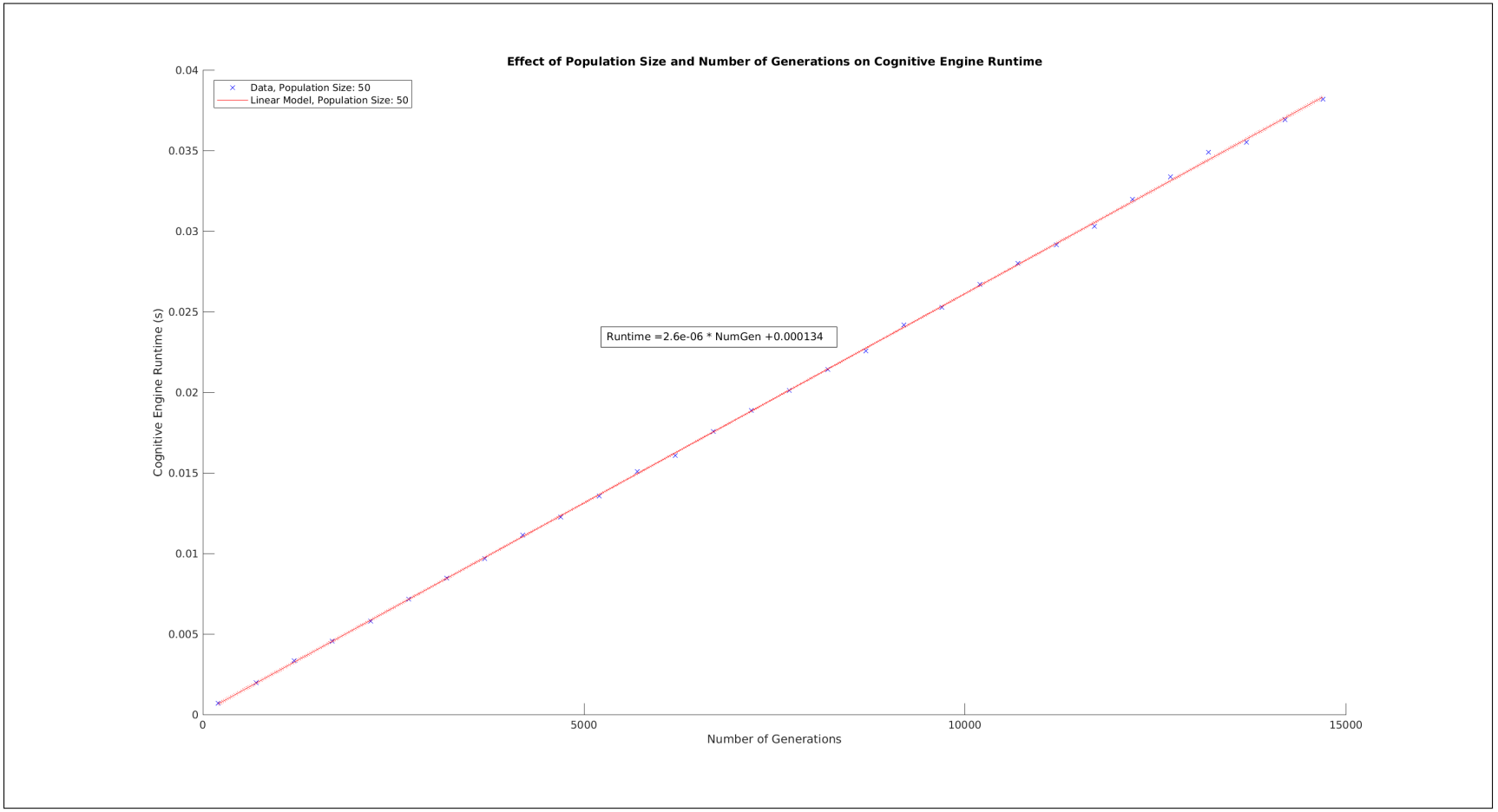


**Figure 26. OMNeT++ engine runtime performance with respect to umber of generations used to evolve solutions**

**Table 12. Coefficient and p-value for OMNeT++ engine runtime growth linear model**

| Term | Value | p-value |
|---|---|---|
| Linear Term Coefficient | $1.345 * 10^{-4}$ | .0157 |
| Intercept | $2.598 * 10^{-6}$ | $5.705 * 10^{-55}$ |

Specifically, the engine can evaluate approximately $385,000$ solutions in a second, using the hardware used in these tests. Given this information and the size of the search space, the amount of time needed to explore a desired percentage of the space could be determined. Conversely, if an interval at which the engine should run is known, the percentage of the search space it could evaluate on each run in looking for good solutions could be determined. In making this determination, it must be decided if transmission medium conditions could change such that previously unfavorable solutions become favorable. This will help inform if a record of the solutions already considered should be retained between engine runs. Additionally, the $385,000$ solutions visited in a second only applies to the hardware used in these experiments. A real-world Unmanned Aerial Vehicle (UAV) swarm is unlikely to have the processing capabilities available during these experiments. As a result, additional tests to evaluate the engine's runtime growth on more representative hardware is warranted.

Ideally, the engine's runtime growth could be contrasted to the improvement in the fitness of its returned solutions by analyzing the effect on the network's latency. The question to be answered is at what point the runtime grows so large or the further improvement in the fitness of solutions found becomes so small as to be prohibitive in using additional generations in evolving good solutions. Unfortunately, the network did not produce the data necessary to perform this analysis due to an issue with its implementation. The fundamental problem is the network detects and discards any transmissions that have bit errors without recording any statistics for them but does not perform any Forward Error Correction (FEC). During the simulation, then,

when the noise floor reaches -10 dBm, at least one bit error is introduced into every transmission, which results in OMNeT++ no longer recording latency statistics for any transmissions. Figure 27 shows a plot of network latency against simulation time that demonstrates the lack of latency information at a simulation time of $1,200$ seconds (corresponding to the noise floor incrementing to $-10$ dBm).

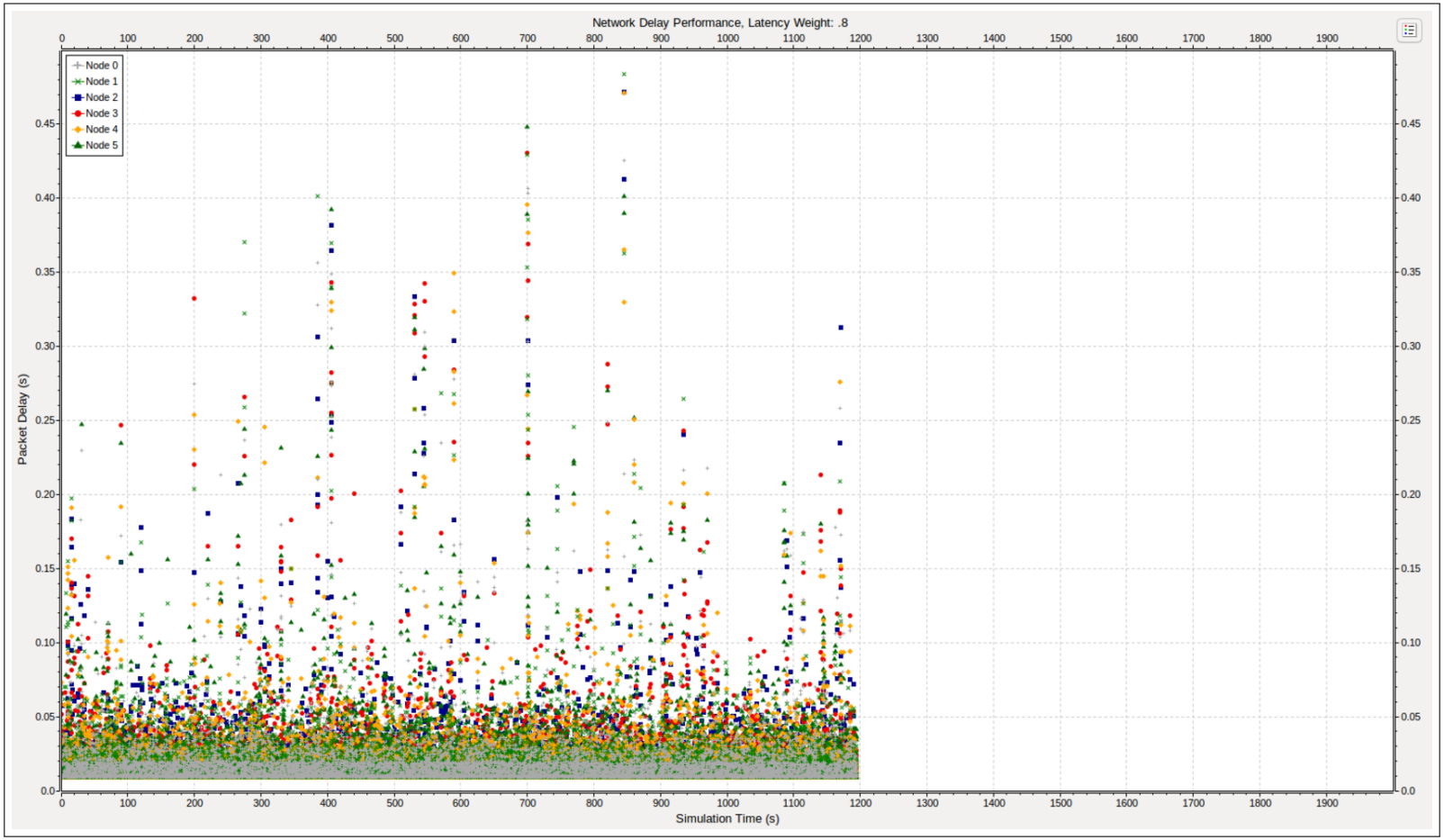


**Figure 27. Plot of network delay against simulation time, demonstrating how OMNeT++ stops recording data part way through simulation**

To combat this, the simulation could be performed with a lower noise floor. However, this does not reflect conditions encountered in the real world and, from (19), leads to both latency and throughput fitnesses of 1. With the low noise floor required to obtain latency data from OMNeT++, the Signal-to-Noise Ratio (SNR) is so high that evaluating Equations 14 and 15 for Bit Error Rate (BER) always returns 0. This corresponds to the ideal performance against which the latency and throughput fitness scores are normalized, leading to a fitness of 1 for both objectives. While higher fitness scores are desirable, reducing the noise floor simply to obtain latency information leads to conditions that are unrealistic and, as a result, relatively uninteresting.

The competing needs to reduce the noise floor to obtain latency information from OMNeT++ and to raise the noise floor to explore interesting and realistic conditions intersect in such a way as to render the OMNeT++ network largely unhelpful in its current state. The addition of FEC encoding and decoding functionality could help combat the bit errors introduced at realistic noise floor levels, allowing the network to record latency statistics under these conditions. Thankfully, even without the FEC functionality, the OMNeT++ tests are not entirely useless. The runtime tests indicate an engine using the same or a similar design and implementation as the one presented here could evaluate a significant portion of the search space in evolving good solutions. This effect becomes even greater if the engine is given more than five seconds to run, as was the case here.

## 5.3 MATLAB Engine Experiments

Due to the issues with obtaining useful information from OMNeT++, the decision was made to re-implement the engine in MATLAB and perform additional tests of both the engine's runtime and performance in terms of the solutions it re-

turned. MATLAB provides a robust signal processing toolbox that includes functionality to perform FEC according to several different algorithms. The MATLAB re-implementation used Bose-Chaudhuri-Hocquenghem (BCH) codes when it used FEC in calculating the BER.

**Engine Runtime Growth and Fitness Performance.**

The experiments again explored the engine's runtime growth, but evaluated this growth with respect to both the number of generations the engine used to evolve solutions and whether or not it used FEC in calculating BER. Further, the MATLAB experiments were also able to contrast the runtime growth with the accompanying change in the fitness of returned solutions. Figures 28 and 29 plot the engine's runtime and returned solution fitness against the number of generations it used in exploring the search space, for non-FEC encoded and FEC encoded transmissions, respectively. Tables 13 and 14 contain the terms for the linear models fit to both the fitness and runtime data as well as their associated p-values. From the tables, the number of generations used beyond 200 did nothing to improve the fitness of the solutions the engine returned, as evidence by the p-value of the linear term coefficients for all fitness models being larger than .05. This is, perhaps, unsurprising. By the time the engine has generated 200 generations, especially given the relatively small search space, it has likely found solutions fairly close to the optimum solution. As a result, generating additional solutions beyond this is not likely to appreciably improve the fitness of the returned solution. Moving forward, 200 generations could be regarded as a maximum on the number of generations used to evolve solutions.

It is also not surprising that once again, increasing the number of generations used to evolve solutions has a positive impact on the engine's runtime, as evidence by the p-values associated with the linear terms for the runtime models in Tables 13

and 14 being much smaller than .05. What is surprising, however, is how much more quickly the runtime grows with the number of generations in the MATLAB implementation as compared to the OMNeT++ implementation. With no FEC encoding, the OMNeT++ implementation could evaluate $\sim 385,000$ more solutions for every additional second of engine runtime. For the MATLAB implementation, however, the runtime grows approximately in direct proportion to the number of generations used to evolve solutions. That is, for every additional generation (i.e., every additional solution considered), the runtime increases by one second, regardless of the noise floor.

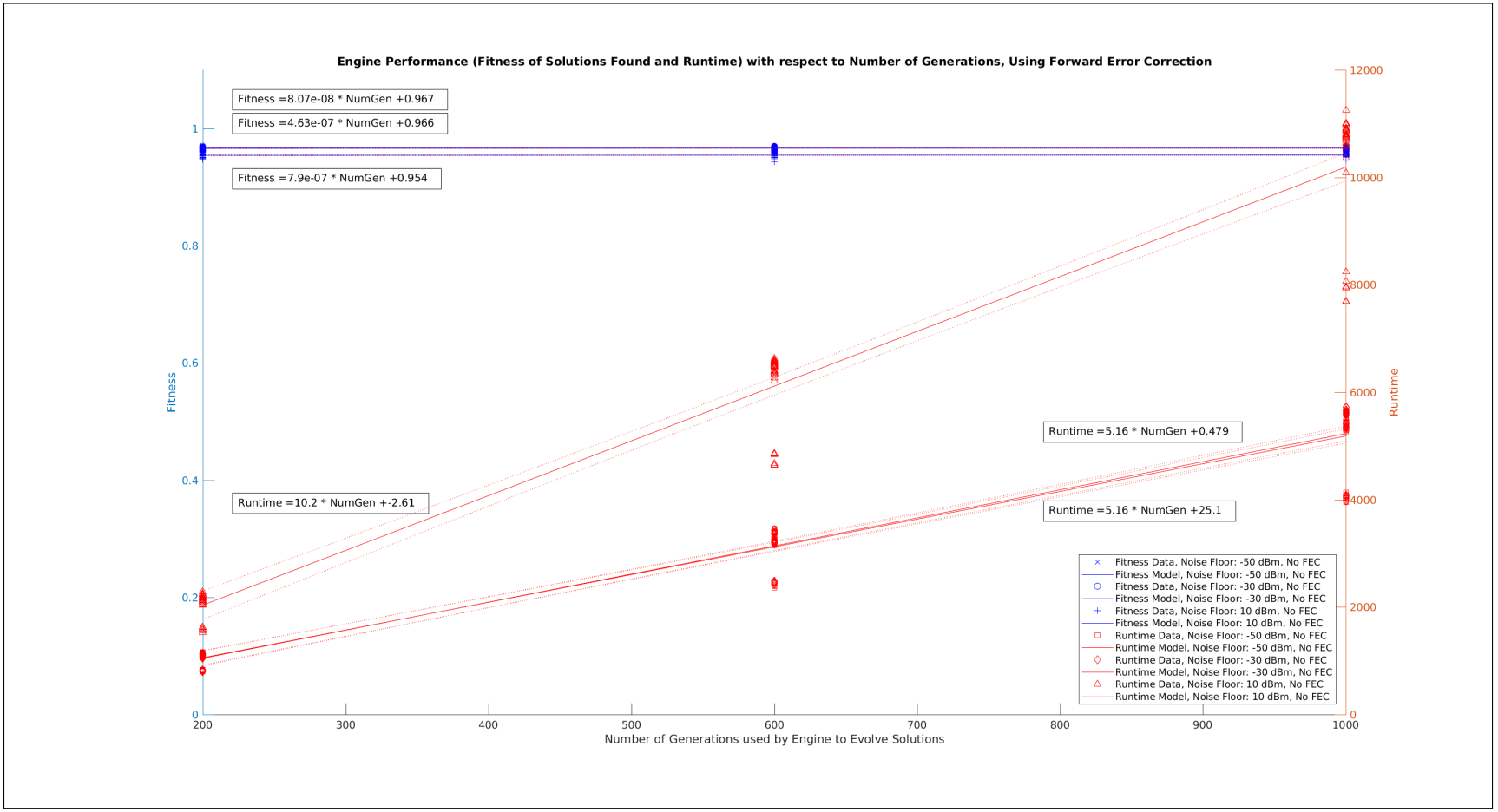


**Figure 28. Growth of engine runtime and fitness of returned solutions with respect to number of generations when no FEC is used**

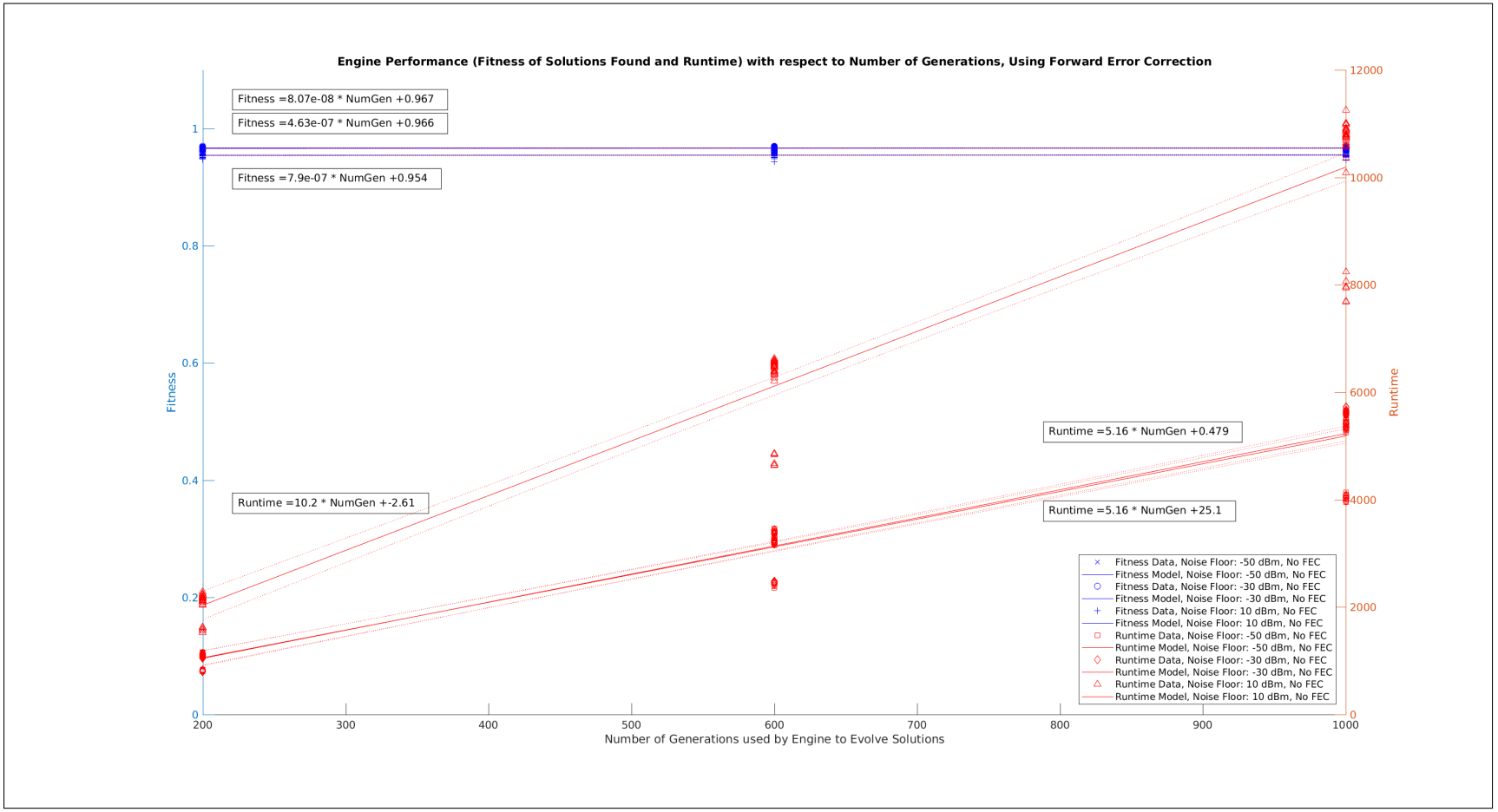


**Figure 29. Growth of engine runtime and fitness of returned solutions with respect to number of generations when FEC is used**

The situation is even worse when the transmissions are FEC encoded. In that case, for noise floors of $-50$ and $-30$ dBm, the runtime grows approximately six and a half times as fast as the number of generations used to find solutions. That is, for every additional generation, the runtime increases by $\sim 6.5$. The performance is worse still for a noise floor of 10 dBm. At that point, the runtime growth with the number of generations is a ratio of approximately $10 - 12$ to 1. That is, for each additional generation, the runtime increases by more than 10 seconds.

While some of the difference in the rate at which the runtimes grow may be explained by differences in how the underlying languages store variables, perform calculations, manage memory, etc., the majority of the difference is due to the manner in which the two implementations compute BER. The OMNeT++ engine computes BER by simply evaluating a function while MATLAB actually simulates bit transmissions, possibly with FEC encoding and decoding. The difference in the runtime growth between the two implementations is an example of the importance of careful planning and decision making in developing a real-world engine for a cognitive UAV swarm. Failure to take the necessary time to plan and design could result in an inefficient engine that can only evaluate a fraction of the solutions a more-efficient engine can consider in the same amount of time. This could result in the engine returning a less-fit solution than it would have found if it were more efficient. In a worst-case scenario, it could fail to return a solution at all within a given time limit. The end result could be a swarm that either cannot adapt to its transmission environment in a timely manner or that exhibits poor latency performance.

One additional item that may be evaluated using this data is whether or not FEC encoding has an appreciable effect on the fitness values of the solutions returned by the engine and the engine's runtime. To do this, a categorical variable encoding whether or not transmissions within that test case were FEC encoded is added to

**Table 13. Coefficient and p-value for linear models for fitness of returned solutions and runtime growth of MATLAB engine using large numbers of generations and no FEC**

| Fitness Linear Model | | | |
|---|---|---|---|
| Noise Floor (dBm) | Term | Value | p-value |
| $-50$ | Linear Term Coefficient | $\sim 8.979 * 10^{-8}$ | $\sim .896$ |
| | Intercept | $\sim .967$ | $\sim 6.393 * 10^{-208}$ |
| $-30$ | Linear Term Coefficient | $\sim 4.267 * 10^{-7}$ | $\sim .541$ |
| | Intercept | $\sim .966$ | $\sim 2.252 * 10^{-207}$ |
| 10 | Linear Term Coefficient | $\sim -3.853 * 10^{-7}$ | $\sim .731$ |
| | Intercept | $\sim .955$ | $\sim 8.194 * 10^{-189}$ |
| | | | |
| Runtime Linear Model | | | |
| Noise Floor (dBm) | Term | Value | p-value |
| $-50$ | Linear Term Coefficient | $\sim 1.002$ | $\sim 8.204 * 10^{-120}$ |
| | Intercept | $\sim 12.365$ | $\sim .000359$ |
| $-30$ | Linear Term Coefficient | $\sim 1.011$ | $\sim 1.212 * 10^{-133}$ |
| | Intercept | $\sim .0891$ | $\sim .970$ |
| 10 | Linear Term Coefficient | $\sim 1.026$ | $\sim 1.749 * 10^{-126}$ |
| | Intercept | $\sim -1.648$ | $\sim .566$ |

**Table 14. Coefficient and p-value for linear models for fitness of returned solutions and runtime growth of MATLAB engine using large numbers of generations and FEC**

| Fitness Linear Model | | | |
|---|---|---|---|
| Noise Floor (dBm) | Term | Value | p-value |
| $-50$ | Linear Term Coefficient | $\sim 8.072 * 10^{-8}$ | $\sim .907$ |
| | Intercept | $\sim .967$ | $\sim 1.114 * 10^{-207}$ |
| $-30$ | Linear Term Coefficient | $\sim 4.631 * 10^{-7}$ | $\sim .505$ |
| | Intercept | $\sim .966$ | $\sim 1.457 * 10^{-207}$ |
| 10 | Linear Term Coefficient | $\sim 7.896 * 10^{-7}$ | $\sim .391$ |
| | Intercept | $\sim .954$ | $\sim 2.656 * 10^{-196}$ |
| | | | |
| Runtime Linear Model | | | |
| Noise Floor (dBm) | Term | Value | p-value |
| $-50$ | Linear Term Coefficient | $\sim 5.160$ | $\sim 1.076 * 10^{-57}$ |
| | Intercept | $\sim 25.075$ | $\sim .780$ |
| $-30$ | Linear Term Coefficient | $\sim 5.229$ | $\sim 3.817 * 10^{-57}$ |
| | Intercept | $\sim .479$ | $\sim .996$ |
| 10 | Linear Term Coefficient | $\sim 10.194$ | $\sim 6.036 * 10^{-58}$ |
| | Intercept | $\sim -2.615$ | $\sim .988$ |

the data table used to produce the linear models previously presented relating fitness and runtime to numbers of generations. A new linear model that takes both number of generations and the new categorical variable into account is fit to the data. The coefficients from this linear model are given in Table 15. From the table, as expected, FEC encoding has a statistically significant impact on runtime but not on the fitness of solutions returned by the engine.

**Engine Runtime Growth and Fitness Performance - Small Numbers of Generations.**

In light of the engine's fitness and runtime performance with numbers of generations larger than 200, specifically the fact that more generations only increased runtime and not fitness, additional experimentation to evaluate its performance with smaller numbers of generations is warranted. In particular, determining the threshold at which the further increase in fitness becomes too small to warrant the additional increase in runtime is of interest. In order to explore this, the engine ran with a number of generations ranging from 5 to 50 in increments of 5. The engine executed 30 times for each number of generations value and its runtime and the fitness value of the returned solution was recorded. A model was then fit to this data to determine how both the fitness and runtime are affected by the number of generations.

It is expected the runtime will again grow linearly as each additional generation requires a constant increase in the required processing. However, it is expected the fitness will increase according to a logarithmic model. This is because very small numbers of generations will not give the engine sufficient time to evolve reasonably fit solutions before being forced to return the most-fit one it has found. However, as the number of generations grows, the fitness will initially increase rapidly as the engine is able to quickly develop better solutions. Then, the improvement will begin

**Table 15. Coefficients and p-values for fitness and runtime linear model with FEC encoding as categorical variable, large numbers of generations**

| Fitness | | |
|---|---|---|
| Term | Value | p-value |
| Number of Generations | $\sim -2.441 * 10^{-7}$ | $\sim .769$ |
| FEC Encoding | $\sim -7.288 * 10^{-5}$ | .893 |
| Intercept | $\sim .962$ | 0 |
| | | |
| Runtime | | |
| Term | Value | p-value |
| Number of Generations | $\sim 3.937$ | $\sim 5.530 * 10^{-64}$ |
| FEC Encoding | $\sim 3512.797$ | $\sim 1.197 * 10^{-99}$ |
| Intercept | $\sim -1750.774$ | $\sim 4.153 * 10^{-27}$ |

to taper off as the engine finds solutions that are close to the optimum such that additional generations will not offer much in the way of fitness improvement.

Figures 30 and 31 show the engine's fitness and runtime performance with respect to the number of generations the engine uses to find solutions for small numbers of generations, without and with FEC encoding, respectively. Additionally, Tables 16 and 17 show the coefficients for the models fit to the data along with their associated p-values. (The figures and tables are analogous to those previously presented for the engine's performance using larger numbers of generations.)

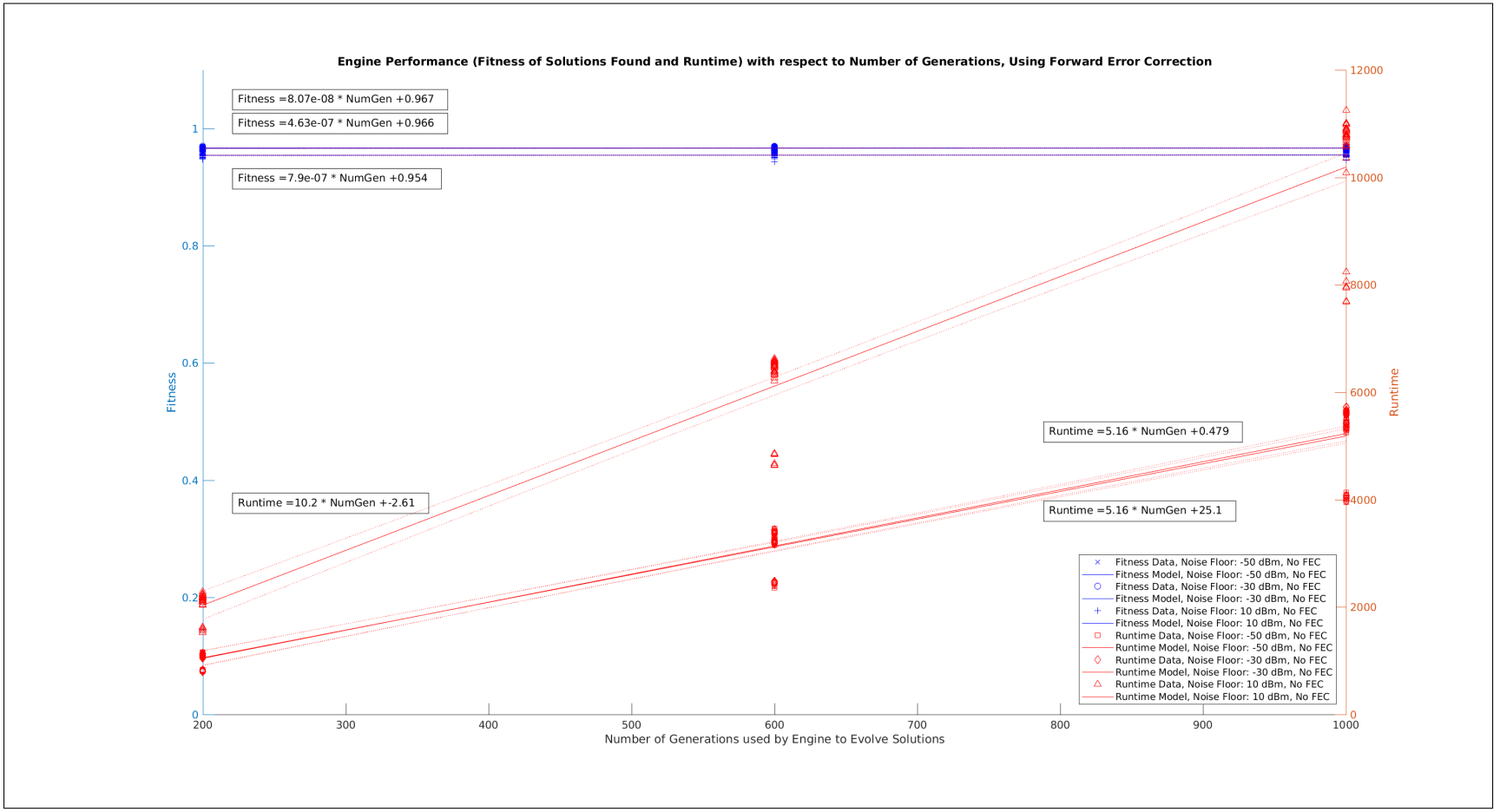


**Figure 30. Growth of engine runtime and fitness of returned solutions with respect to number of generations when no FEC is used**

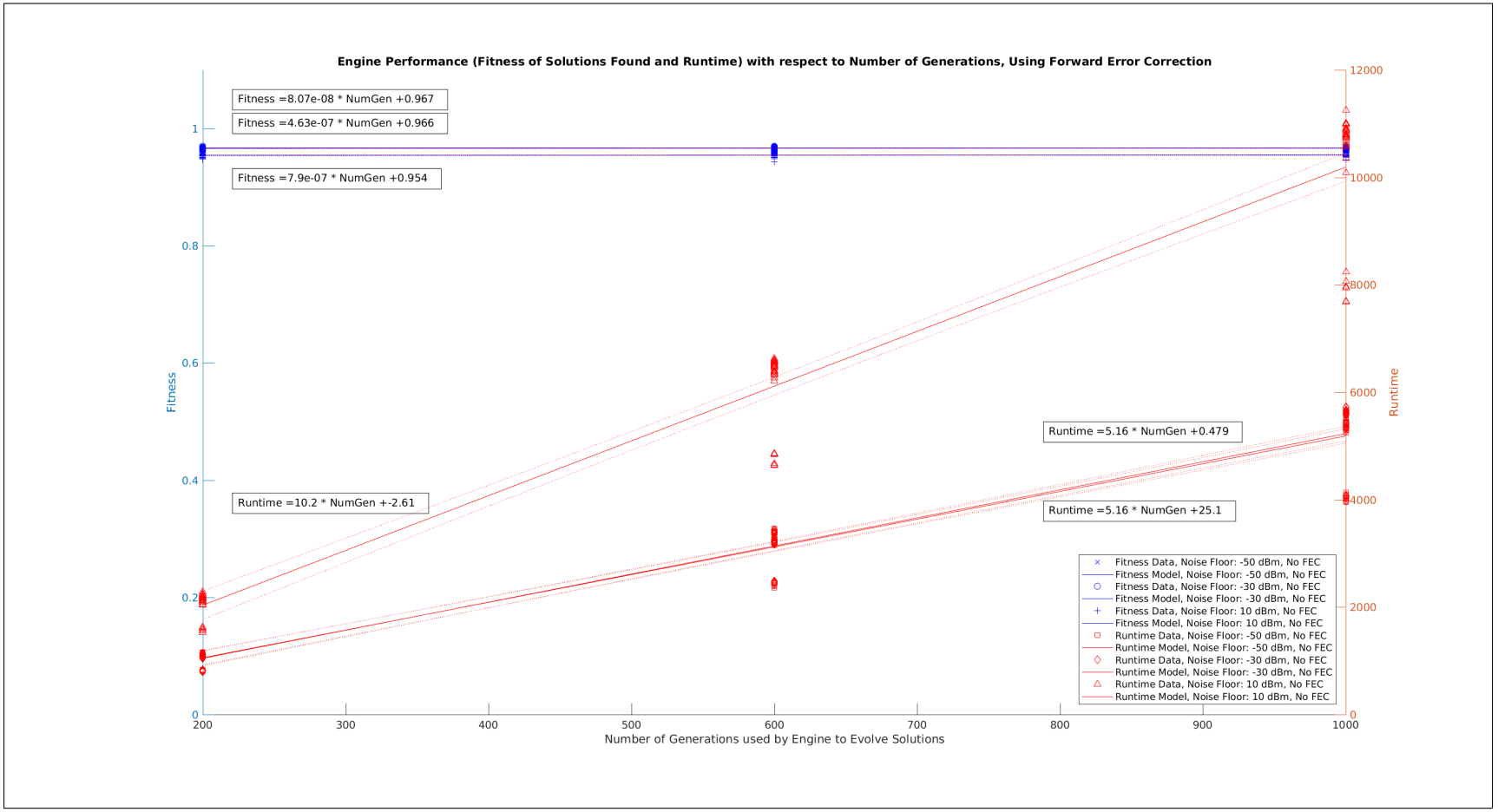


**Figure 31. Growth of engine runtime and fitness of returned solutions with respect to number of generations when FEC is used**

Several things are immediately noticeable from the data. First, as expected, the runtime again increases linearly with the number of generations. Comparing Table 13 with Table 16 and Table 14 with Table 17, the rates at which the engine's runtime increases with respect to the number of generations, taking into account whether or not the data is FEC encoded, are similar.

The second immediately noticeable item is that the engine's fitness performance does not increase according to a logarithmic model as expected. In fact, as previously, the linear models fit to the data indicate the number of generations does not have a statistically significant impact on the fitness value of the solutions returned by the engine. While unexpected, there are several reasons why this may be the case. The first possibility is the engine always finds a reasonably good solution as it initializes the population. As a result, even if no solutions that are relatively close to the optimum are considered as the engine runs, it has this good solution in the population it can return. Given the relatively small size of the search space, this is not outside the realm of possibility. Another possibility is the engine always generates one solution that is relatively close to optimum as it runs. While again not outside the realm of possibility, due to the small numbers of generations involved, this is harder to accept.

The more likely scenario is that the fitness function does not adequately distribute the fitness values of the possible solutions in the search space between 0 and 1. That is, the fitness function may be assigning relatively high values ($> .9$) to a disproportionately large number of possible solutions and smaller fitness values to a relatively few number of solutions. The issue with this possibility is that it could be masking solutions that would actually produce poor latency performance in a real-world network as a solution that would produce acceptable performance. As MATLAB produces no actual latency information, the fitness values of returned solutions must be taken as an analog of the latency itself. However, the realized performance will

**Table 16. Coefficient and p-value for linear models for fitness of returned solutions and runtime growth of MATLAB engine using small numbers of generations and no FEC**

| Fitness Linear Model | | | |
|---|---|---|---|
| Noise Floor (dBm) | Term | Value | p-value |
| $-50$ | Linear Term Coefficient | $\sim 2.095 * 10^{-6}$ | $\sim .807$ |
| | Intercept | $\sim .967$ | 0 |
| $-30$ | Linear Term Coefficient | $\sim -5.812 * 10^{-6}$ | $\sim .482$ |
| | Intercept | $\sim .967$ | 0 |
| 10 | Linear Term Coefficient | $\sim -1.423 * 10^{-5}$ | $\sim .312$ |
| | Intercept | $\sim .955$ | 0 |
| | | | |
| Runtime Linear Model | | | |
| Noise Floor (dBm) | Term | Value | p-value |
| $-50$ | Linear Term Coefficient | $\sim 1.012$ | 0 |
| | Intercept | $\sim -.727$ | $\sim .1.612 * 10^{-6}$ |
| $-30$ | Linear Term Coefficient | $\sim .998$ | 0 |
| | Intercept | $\sim -.497$ | $\sim .000670$ |
| 10 | Linear Term Coefficient | $\sim 1.018$ | 0 |
| | Intercept | $\sim -.894$ | $\sim 4.096 * 10^{-10}$ |

**Table 17. Coefficient and p-value for linear models for fitness of returned solutions and runtime growth of MATLAB engine using small numbers of generations and FEC**

| Fitness Linear Model | | | |
|---|---|---|---|
| Noise Floor (dBm) | Term | Value | p-value |
| $-50$ | Linear Term Coefficient | $\sim 2.095 * 10^{-6}$ | $\sim .807$ |
| | Intercept | $\sim .967$ | 0 |
| $-30$ | Linear Term Coefficient | $\sim 3.901 * 10^{-6}$ | $\sim .644$ |
| | Intercept | $\sim .967$ | 0 |
| 10 | Linear Term Coefficient | $\sim -7.749 * 10^{-6}$ | $\sim .548$ |
| | Intercept | $\sim .954$ | 0 |
| | | | |
| Runtime Linear Model | | | |
| Noise Floor (dBm) | Term | Value | p-value |
| $-50$ | Linear Term Coefficient | $\sim 6.549$ | 0 |
| | Intercept | $\sim .229$ | $\sim .586$ |
| $-30$ | Linear Term Coefficient | $\sim 6.556$ | 0 |
| | Intercept | $\sim -.386$ | $\sim .346$ |
| 10 | Linear Term Coefficient | $\sim 12.891$ | 0 |
| | Intercept | $\sim -.859$ | $\sim .638$ |

only be as good as the fitness function on which the engine is built. As a result, if the fitness function exhibits poor performance (e.g., assigns high fitness values to a disproportionately large number of solutions such that solutions that produce poor performance are given high values), the network's performance will likely be poor as well. Unfortunately, with no approach to gleaning latency information from MATLAB, there is no way to directly test the fitness function's quality. In order to do this, the best approach would likely be to implement FEC in the OMNeT++ network and test various solutions given both high and low fitness values by the fitness function. From these tests, whether or not the fitness function assigns high fitness values only to those solutions that produce acceptable latency fitness could be determined.

As with the fitness of solutions the engine returns and its runtime with large numbers of generations, the effect of FEC encoding on these performance metrics may be evaluated. To do this, a categorical variable is again added to the data used to produce the linear models relating fitness and runtime to numbers of generations. The coefficients for the new linear models relating fitness and runtime to number of generations and FEC encoding are given in Table 18. From the table, as expected, FEC encoding has a statistically significant impact on runtime but not on the fitness of solutions returned by the engine.

**Fitness Variance as Parameters Change from Optimal Settings.**

To start evaluating how well the fitness function distributes fitness scores, how they change as each parameter deviates from its optimum setting may be considered. To start this investigation, the ideal setting for each parameter for each noise floor ($-50$, $-30$ and 10 dBm) and when the transmissions are and are not FEC encoded are needed. (In general, these values cannot be found as the search space is too large to exhaustively search for the best result. The search space under consideration here

**Table 18. Coefficients and p-values for fitness and runtime linear model with FEC encoding as categorical variable, small numbers of generations**

| Fitness | | |
|---|---|---|
| Term | Value | p-value |
| Number of Generations | $\sim -3.757 * 10^{-6}$ | $\sim .723$ |
| FEC Encoding | $\sim . - .000166$ | .586 |
| Intercept | $\sim .963$ | 0 |
| | | |
| Runtime | | |
| Term | Value | p-value |
| Number of Generations | $\sim 4.837$ | $\sim 1.131 * 10^{-199}$ |
| FEC Encoding | $\sim 210.918$ | 0 |
| Intercept | $\sim -105.982$ | $\sim 1.593 * 10^{-95}$ |

is a special case, small enough that each possible solution may be considered. In order to take better advantage of the power of genetic algorithms, the search space should be larger. Recommendations for increasing the number of possible solutions are given in Chapter VI.) These values are given in Table 19. Once the optimum value for each parameter is known, how the fitness values change as each deviates from this setting may be evaluated by fixing three of the four parameters and allowing the fourth to take on each of its possible values. The fitness of each resulting parameter set may be evaluated and the results plotted to graphically depict the effect on solution fitness. These plots are given below.

**Table 19. Results for exhaustive search for most-fit solutions depending on noise floor and FEC encoding**

| Noise Floor (dBm) | FEC Encoded? | Modulation Scheme | Modulation Order | Power (dBm) | Bandwidth (MHz) | Fitness Value |
|---|---|---|---|---|---|---|
| -50 | No | PSK | 3 | 4 | 1 | .9715 |
| -50 | Yes | PSK | 3 | 4 | 1 | .9715 |
| -30 | No | PSK | 3 | 4 | 1 | .9715 |
| -30 | Yes | PSK | 3 | 4 | 1 | .9715 |
| 10 | No | PSK | 2 | 25 | 1 | .9598 |
| 10 | Yes | PSK | 2 | 19 | 1 | .9627 |

**Modulation Scheme.**

From Table 19, the optimum modulation scheme for all noise floors both with and without FEC encoding is Phase Shifft Keying (PSK). However, Figure 32 shows that using Quadrature Amplitude Modulatio (QAM) rather than PSK, assuming all other parameters are at their optimum setting, does not appreciably decrease solution fitness. This is not overly surprising considering the processing that occurs with QAM and PSK modulation. As its name implies, PSK involves modulating the phase of the carrier signal to encode the data. QAM derives its name from the fact it involves modulating the amplitudes of two signals that are at the same frequency but 90 degrees out-of-phase (in quadrature) with respect to each other. Mathematically, this is equivalent to modulating both the phase and amplitude of a single carrier signal. As a result, when using a relatively small constellation size as discussed in the next section, both modulation schemes are robust against the interference encountered at the noise levels considered here. Thus, the solution fitness value remains relatively high regardless of the modulation scheme used.

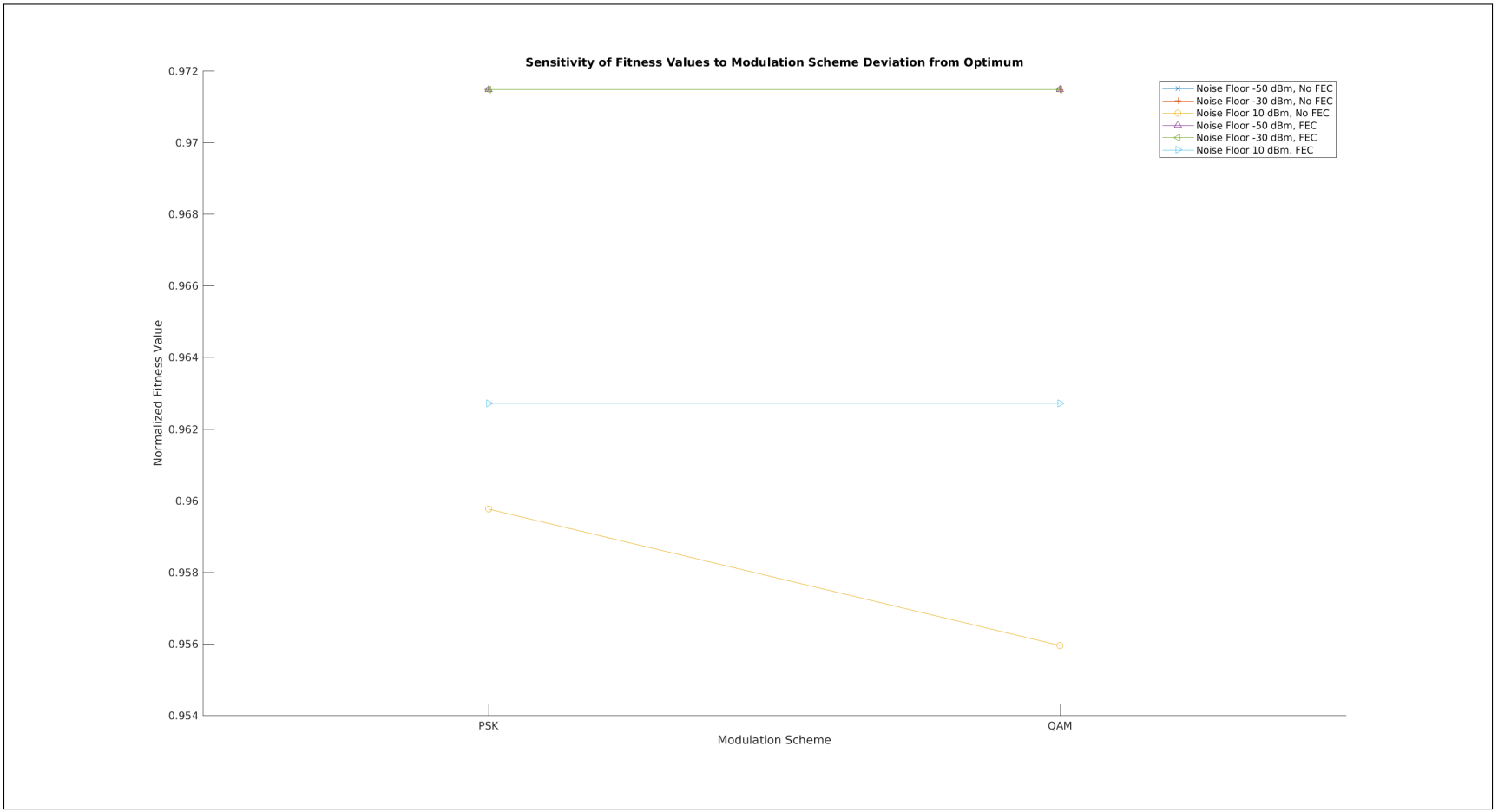


**Figure 32. Sensitivity of solution fitness value as modulation scheme deviates from optimum value**

**Modulation Order.**

From Table 19, the optimum constellation size for a noise floor of $-50$ and $-30$ dBm is 8 while for a noise floor of 10 dBm it is 4 constellation points. In general, then, as the noise floor increases, the ideal constellation size decreases, as expected. It may perhaps be surprising the optimum constellation size for the $-50$ and $-30$ dBm noise floors is not lower; the interference caused by such low noise levels is certainly not high enough to render a larger constellation size unusable. The reason the optimum size is 8 for these noise floors is likely due to the increased processing and thus increased power consumption involved in using a higher modulation order. The fact the fitness function takes power consumption into account is the likely reason the optimum constellation size is lower than what it might otherwise have been.

Figure 33 shows how solution fitness values deviate from their optimum as the constellation size varies. Unlike modulation scheme, modulation order has a significant impact on solution fitness. For a noise floor of $-50$ dBm, the interference is low enough a constellation with 256 points could be used and still maintain high fitness. However, as the noise floor rises to even $-30$ dBm, the maximum size the constellation may have and still facilitate reliable communication becomes constrained. Using FEC encoding with this noise floor allows a constellation with 128 symbols to be used whereas without FEC encoding, the constellation size is constrained to 64 symbols. As expected, then, FEC encoding allows the receiver to detect and correct bit errors, allowing for a larger constellation size and thus greater throughput. However, this comes at the cost of additional processing and thus greater power consumption. The situation is even more pronounced for a noise floor of 10 dBm. In that case, the interference is so large that a constellation size greater than 4 symbols, even with FEC encoding, results in the receiver not being able to correctly interpret the transmitted data. (It should be noted a higher transmission power could allow for a larger

constellation size to be used and could show a difference in maximum constellation size between FEC and non-FEC encoded data. However, as part of these tests, the transmission power was held at its optimum value of 19 and 25 dBm for a noise floor of 10 dBm with and without FEC encoding, respectively.)

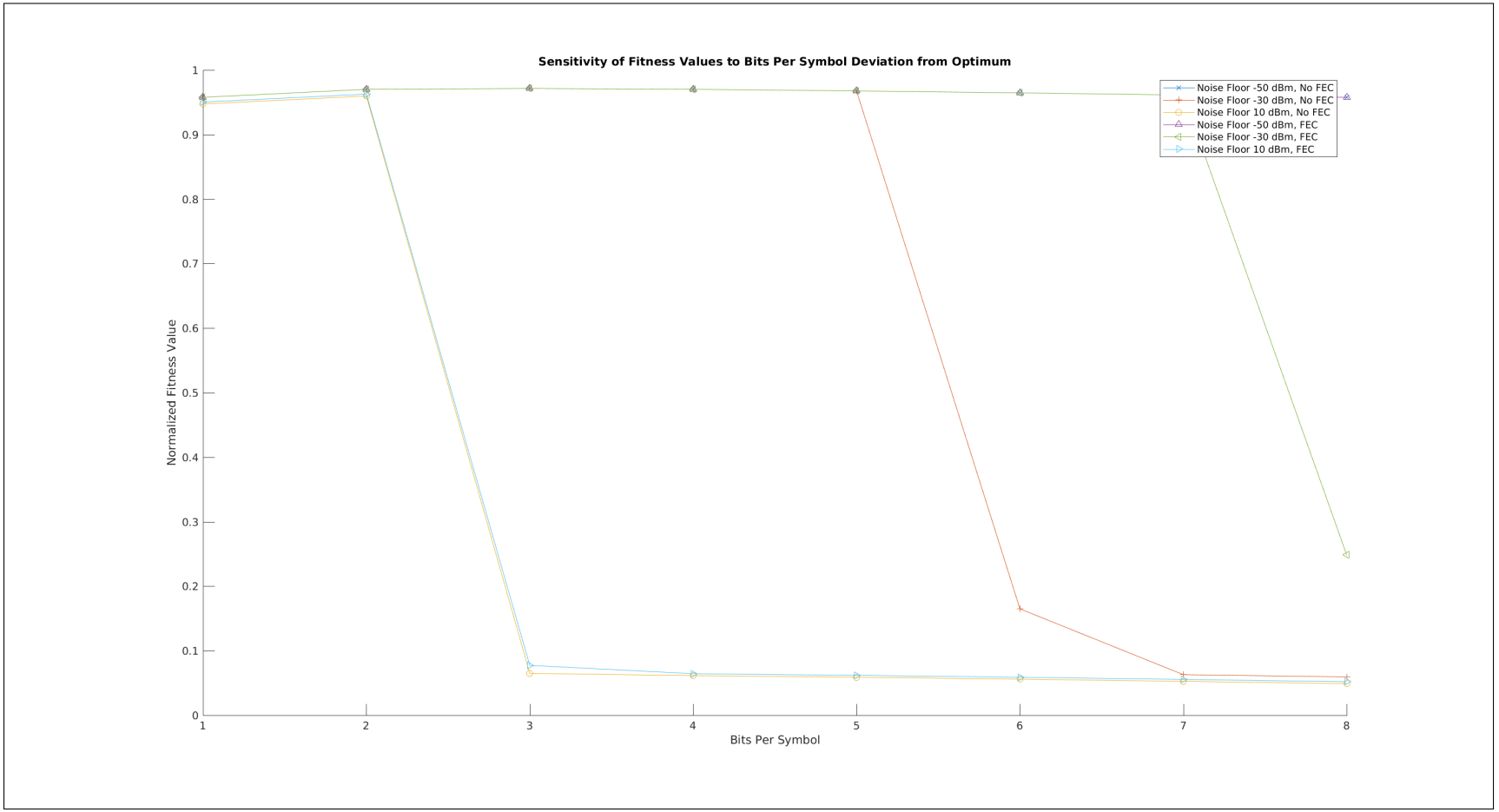


**Figure 33. Sensitivity of solution fitness value as bits per second deviates from optimum value**

**Power Setting.**

The optimum power setting for both the $-50$ and $-30$ dBm cases, from Table 19, is 4 dBm. For a noise floor of 10 dBm, it is 19 and 25 dBm with and without FEC encoding, respectively. These trends are as expected. In the first two cases, the noise floors are low enough they do not cause a significant amount of interference, even when the transmitters use a low transmit power. As a result, the engine reduces power consumption a much as it can in accordance with its fitness function while still maintaining a reliable communication channel. The situation is different for a noise floor of 10 dBm. In that case, the interference has rise enough the engine decides the transmit power must increase to allow for communication. Here again, the effect of FEC encoding may be observed. Whereas it did not impact the modulation order with which transmissions could be sent with a noise floor of 10 dBm, it does affect the transmission power required. From Figure 34 which shows how solution fitness values change with transmission power, FEC encoding allows the transmit power to be reduced for a noise floor of 10 dBm from what was required for reliable communication without FEC encoding.

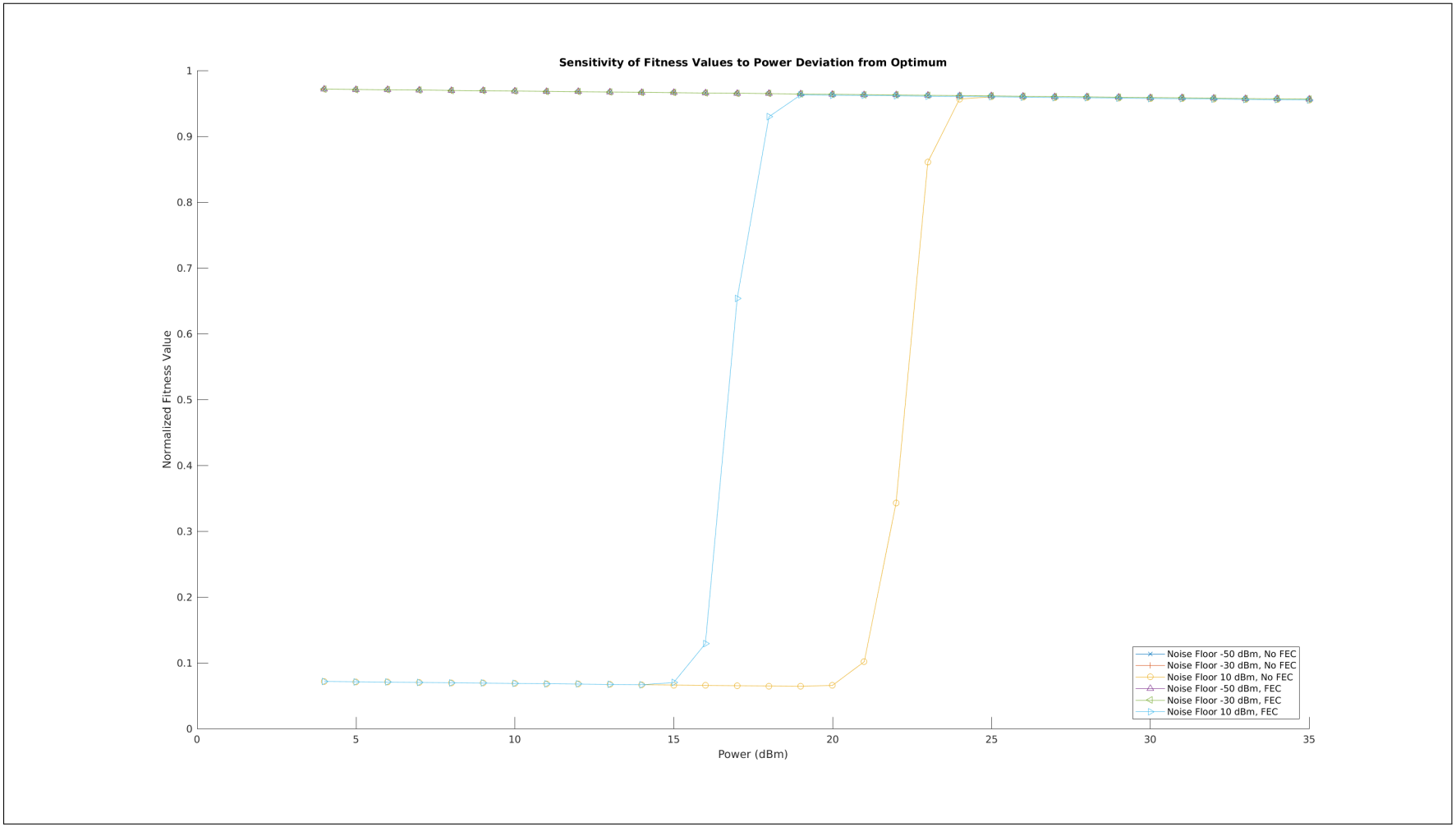


**Figure 34. Sensitivity of solution fitness value as power deviates from optimum value**

**Bandwidth.**

Table 19 shows the optimum bandwidth for each noise floor with and without FEC encoding is 1 megahertz while Figure 35 shows how solution fitness varies with bandwidth. From the figure, the fitness approximately linearly decreases in all cases as bandwidth increases. This is likely because the power consumption component of the fitness function penalizes higher bandwidth settings due to the additional processing required. Further, increasing the bandwidth does not affect the throughput or latency fitness because, in order for this to happen, the bitrate must change with the bandwidth. However, the engine developed and presented here does not do this. As a result, increasing the bandwidth linearly decreases fitness due to the increased power consumption with no change in the throughput or latency fitness.

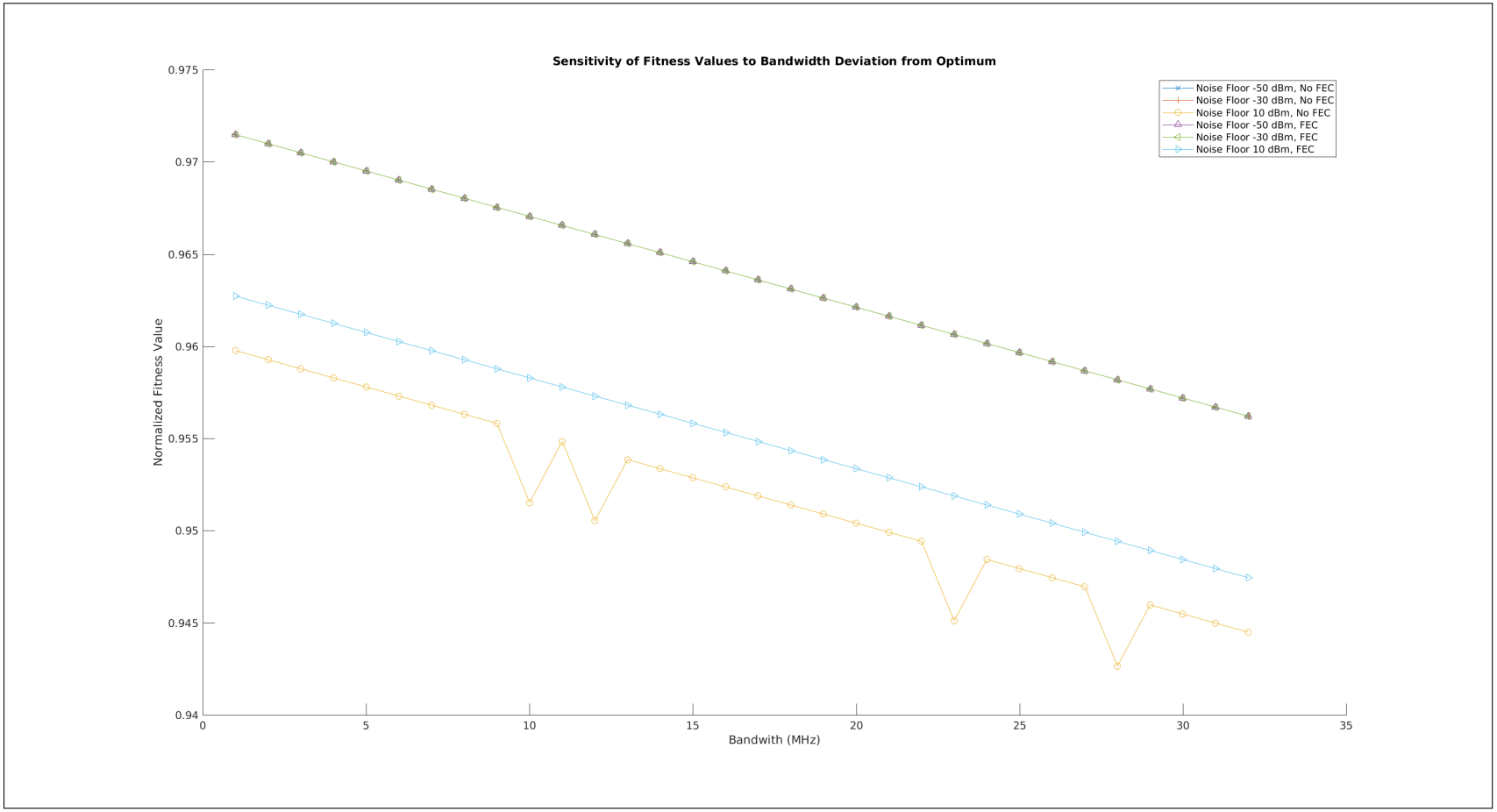


**Figure 35. Sensitivity of solution fitness value as bandwidth deviates from optimum value**

### Frequency with which Engine Returns Each Parameter Value.

The final item of interest is to investigate the probability with which the engine returns each possible value for each parameter. To do so, the engine was run 90 times at each noise floor both with and without FEC encoding and the solution the engine returns each time was recorded. The number of times each value for each parameter occured in these 90 solutions was counted and frequency plots for each parameter were constructed. (These plots are included in Appendix D.) This information may be evaluated in light of the optimum solutions and how solution fitness changes as the parameters deviate from these optimum values as explored in the previous section to gain insight into how likely the engine is to return a solution with low fitness.

#### Modulation Scheme.

Figures 38 and 39 indicate a fairly even split between how often the engine returned a solution using PSK vice QAM across all noise floors, both with and without FEC encoding. From the preceding section, however, using QAM as opposed to PSK did not appreciably diminish solution fitness. As a result, although the engine may be expected to return a solution using the non-optimum modulation scheme almost half of the time, it is not expected the negative impact on the engine's fitness performance is significant.

#### Constellation Size.

Figures 40 and 41 give the plots for the frequency with which the engine returns each possible value for constellation size. For a noise floor of $-50$ dBm, the engine returns a constellation size of 4 with the greatest frequency, a constellation size of 8 and 16 with slightly smaller frequency and a larger constellation size a small fraction of the time. However, as solution fitness is relatively unaffected by constellation size

for this noise floor, the variance in the constellation size the engine returns is not a cause for concern as it will not appreciably diminish the fitness performance.

For a noise floor of $-30$, the engine returns constellation sizes of 4, 8 or 16 with similar frequencies. It also returns a larger constellation size a small portion of the time. With FEC encoding, this is not an issue; the constellation size must be 256 before solution fitness is negatively affected and these results do not indicate the engine ever returns such a large constellation. However, without FEC encoding, fitness is severely impacted with a constellation size of 64 and these results indicate the engine returns this constellation size for a noise floor of $-30$ dBm without FEC Encoding a small percentage of the time. In the real world, this would likely translate into communication within the network being temporarily interrupted until the engine runs again and updates the parameters to new values. While link interruption due to unforeseen circumstances is always a risk, having known engine behavior that could result in such an interruption is unacceptable. Therefore, additional work to ensure the engine never returns an untenable solution needs to be performed. This work could possibly look to revise the fitness function or explore using additional generations to ensure the engine always finds at least one tenable solution so it does cause link interruption.

The situation is even worse for a noise floor of 10 dBm. Without FEC encoding, the engine returns a constellation size that results in link interruption almost 30% of the time. With FEC encoding, this still happens almost 20% of the time. As with the $-30$ dBm case, especially considering a noise floor of 10 dBm is much closer to what real-world networks will experience, this behavior is unacceptable. This further underscores the need for future work in refining the engine to ensure its operation does not lead to link interruption.

**Power.**

Figures 42 and 43 give the engine's frequency plots for power. As the results in the previous section indicate solution fitness is relatively unaffected by power setting for the $-50$ and $-30$ dBm noise floors, there is no concern the engine will cause link interruption in these cases. However, it is interesting to note the frequency with which the engine returns a power setting higher than the optimum value of 1 dBm. This translates into the network consuming more power than necessary and is likely due to the relatively low weight the power component of the fitness function was given. Additional work looking at assigning different weight to the three fitness function components could be performed to see if this behavior could be diminished by adjusting the weights. However, care would need to be taken that altering the weights does not either lead to the engine failing to return an acceptable solution when one exists or returning unacceptable solutions. Both with and without FEC encoding, the engine always returns a power setting of 24 dBm or higher for a noise floor of 10 dBm. While this does not lead to issues with link interruption, similar to the $-50$ ad $-30$ dBm cases, this could lead to unnecessarily high power consumption. This further underscores the need for additional work to tune the engine to ensure it always returns an acceptable solution while minimizing resource use.

**Bandwidth.**

As the previous section discussed, solution fitness is largely unaffected by bandwidth changes. Thus, the frequencies with which the engine returns different bandwidth values as shown in Figures 36 and 37 are not a cause for concern in term of link interruption or poor fitness performance. However, one direction in which the engine could be taken is to allow it to alter the bitrate in the network when it adjusts the nodes' bandwidth. If this is implemented in the network, it is expected changing

the bandwidth (and bitrate) would have a greater effect on solution fitness. In that case, the frequencies with which the engine returns each bandwidth value could take on greater importance.

# VI. Conclusions and Recommendations

Based on the results presented in the previous chapter, there are several possible directions in which this Cognitive Engine (CE) research could move in the future.

**Implementing FEC Encoding in OMNeT++ Network.**

The most important future effort is arguably to implement Forward Error Correction (FEC) in the OMNeT++ network, or otherwise develop a test network from which latency performance data may be obtained for realistic operating scenarios. This will be important in investigating the quality of the fitness function in terms of the latency performance realized using solutions to which it assigns high fitness values. The INET framework provides the "APSK Layered Transmitter" and analogous receiver module types which provide a skeleton into which submodules that implement the desired functionality may be placed. Further, these transmitter and receiver types could be placed into existing radio types. However, a solution using another network simulation framework is also possible.

**Evaluating Distribution of Fitness Values Assigned by Fitness Function.**

As discussed in Chapter V, the fitness performance exhibited by the engine may be an indication of its efficiency in finding highly fit solutions. However, a more likely scenario is that the fitness function assigns a disproportionately large number of solutions high fitness values. As a result, work needs to be done to assess the distribution of fitness scores across the search space. Additionally, the likelihood with which the fitness function assigns a high fitness value to a solution that leads to poor network latency performance needs to be assessed. It may be a slight increase in this likelihood is acceptable if it leads to a greater increase in the likelihood of the engine finding a highly fit solution that yields low network latencies. Conversely, it may be

that the possibility of the engine returning a solution that leads to poor performance is unacceptable and thus this probability must be minimized. An OMNeT++ network that implements FEC or another network from which actual latency information may be obtain, as discussed in the previous section, will enable this work.

**Testing the Engine with Additional Noise Floors.**

Another possible explanation for the consistently high engine performance is that the low noise levels used in the experiments conducted here allow for relatively error-free communication (leading to low Bit Error Rates (BERs) and high fitness values in the MATLAB simulation), regardless of the parameter settings. As a result, another possible direction for future work would be to repeat these experiments, or variations thereof, using additional, high noise floors. These experiments should focus on determining whether or not the high probability of the engine finding a highly fit solution as with the experiments presented here carries over to high noise floors. If not, increased interference may the engine to begin discerning between truly fit solutions and those that would likely not lead to acceptable performance in real-world networks.

**Allowing Engine to Alter Bitrate with Bandwidth.**

As discussed in Chapter V, altering the network's bandwidth without also allowing the engine to alter the bitrate negates the effect of changing the bandwidth. This resulted in variations in the bandwidth having little effect on solution fitness. As a result, future work should be conducted to allow the engine to alter both the network bandwidth and bitrate. Then, additional tests to evaluate how changes in the network's bandwidth and bitrate affect the realized latency performance should be conducted.

**Recommendations regarding the use of CORNET for Cognitive Network Tests.**

While it would be irresponsible to suggest not using CORNET in any future testing efforts, care is warranted in its use. In particular, a thorough understanding of CORNET's construction, what it has been optimized to measure in term of performance and what it not adept at measuring is needed. For example, CORNET's original design focused on facilitating throughput measurements and using these in cognitive engine designs. The reason for this is straightforward; in most cases, when talking about network performance, throughput is the metric of interest. Indeed, latency has only recently begun to gain attention as a Quality of Service (QoS) metric. As a result, while CORNET inherently supports latency measurements through the timestamp and message number vectors it produces, a different, more reliable method for measuring latency is needed if CORNET is to be used in future research where latency is the QoS metric of interest. Perhaps the most efficient path towards developing such a method is through a dialog with the CORNET administrators to gain additional insight into the system's strengths, weaknesses, capabilities and the best way to go about measuring latency.

**FEC Encoding and Number of Generations Trade-off Analysis.**

As discussed in Chapter V, the experimental data indicated that, if the Signal-to-Noise Ratio (SNR) is low enough, FEC encoding the data may result in an increased engine runtime without a similar increase in the fitness of returned solutions of sufficient magnitude to warrant the runtime penalty. Similarly, the data also indicated for even a relatively small number of generations, the increase in the fitness of resulting solutions may not be sufficiently greater than those returned by the engine using a smaller number of generations as to warrant the increased runtime. Additional work

is needed to study the trade-off between FEC encoding, the number of generations the engine uses in developing solutions and their affect on runtime in order to determine when FEC encoding should and should not be used and how to determine the number of generations that should be used. This work should include what data is necessary to making these decision and how to code them into the engine itself.

**Increasing the Engine's Search Space to take Greater Advantage of the Genetic Algorithm.**

As discussed in previous chapters, the true power of genetic algorithms lies in their ability to quickly find "good" solutions in a large search space. Admittedly, the search space used here is small enough that it cannot truly take advantage of the genetic algorithm paradigm. However, the search space used may be readily extended such that the resulting search space is large enough to take advantage of the genetic algorithmic paradigm. For example, the search space used here has

$$\text{Search Space Size} = 2 * 8 * 32 * 32 = 16,384 \text{ solutions} \tag{20}$$

To determine this, the number of possible values for each of the parameters made available to the engine are multiplied together. By adding just three additional variables with 32 possible values each, the search space grows to

$$\text{New Search Space Size} = 2 * 8 * 32 * 32 * 32 * 32 * 32 = 536,870,912 \text{ solutions} \tag{21}$$

It should be noted this growth in the search space could come from a greater number of additional parameters with a smaller number of possible values each. It doesn't have to be just three parameters or parameters with 32 possible values. However, by

increasing the size of the search space, additional, possibly more-fit solutions become available and the engine may more readily take advantage of the power offered by the genetic algorithm.

**Investigating the Effect of Latency Weight.**

Beyond placing additional parameters in the CE, how the weight given to the latency component of the fitness function affects the network's performance could be explored. Further, how the weight left to be assigned after assigning a certain value to the latency is divided amongst the throughput and power fitness objectives could also be explored.

**Repeating Runtime Experiments with Representative Hardware.**

As discussed in Chapter 5, while general conclusions may be drawn from the runtime data presented, the actual values are only valid for the hardware on which the tests were run. In particular, while it may be expected a reasonable amount of the search space may still be explored by a real-world CE on an Unmanned Aerial Vehicle (UAV), hardware difference may constrain the portion of the search space that may be explored in a given amount of time or may increase the time required to investigate a set portion of the search space. As a result, runtime tests need to be repeated using representative hardware to what would be found on an actual UAV to determine how the limitations imposed by the hardware affect the CE's performance.

**Effect of Retaining Population Between Engine Runs.**

For the tests conducted here, the CE reinitialized its population every time it ran. This prevented previously found "good" solutions that were no longer acceptable due to changes in the transmission environment from adversely affecting the solutions the

engine considered in subsequent runs. However, this increases the engine's runtime. Additionally, for transmission mediums that do not change quickly, as would likely be encountered by operational UAV swarms, maintaining knowledge of previously found "good" solutions would likely help improve the engine's fitness performance. In light of this, future work needs to be performed to implement some form of "genetic memory" between CE runs.

# Bibliography


1. A. C. Gibson. "SecAF: The first interview". [Online]. Available: http://www.af.mil/News/Article-Display/Article/1199661/secaf-the-first-interview/.

2. D. L. James and M. A. W. III, "America's Air Force: A Call to the Future," Office of the Secretary of the Air Force, Report, 2014.

3. "DARPA Software Defined Radio (SDR) Hackfest Selects Teams to Explore Cyber-Physical Intersection of SDR and Drone Technology". [Online]. Available: https://www.darpa.mil/news-events/2017-10-16.

4. M. R. Endsley, "Autonomous Horizons: System Autonomy in the Air Force - A Path to the Future," Office of the Chief Scientist, Report, 2015.

5. C. W. Bostian and A. R. Young, "The Application of Cognitive Radio to Coordinated Unmanned Aerial Vehicle (UAV) Missions," Virginia Polytechnic Institute and State University, Report, 2011.

6. K. Osborn. "Swarming Mini-Drones: Inside the Pentagon's Plan to Overwhelm Russian and Chinese Air Defenses". [Online]. Available: http://nationalinterest.org/blog/the-buzz/swarming-mini-drones-inside-the-pentagons-plan-overwhelm-16135.

7. A. A. Khan, M. H. Rehmani, and M. Reisslein, "Cognitive radio for smart grids: Survey of architectures, spectrum sensing mechanisms, and networking protocols," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 860–898, 2016.

8. F. Ge, Q. Chen, Y. Wang, C. W. Bostian, T. W. Rondeau, and B. Le, "Cognitive radio: From spectrum sharing to adaptive learning and reconfiguration," in *IEEE Aerospace Conference*, March 2008, pp. 1–10.

9. J. Mitola, "Cognitive radio architecture evolution," *Proceedings of the IEEE*, vol. 97, no. 4, pp. 626–641, April 2009.

10. I. Mitola, Joseph, "Cognitive radio. an integrated agent architecture for software defined radio," Ph.D. dissertation, Virginia Polytechnic Institute and State University, 2000.

11. J. Mitola and G. Q. Maguire, "Cognitive radio: making software radios more personal," *IEEE Personal Communications*, vol. 6, no. 4, pp. 13–18, Aug 1999.

12. V. B. Alluri, J. R. Heath, and M. Lhamon, "A new multichannel, coherent amplitude modulated, time-division multiplexed, software-defined radio receiver architecture, and field-programmable-gate-array technology implementation," *IEEE Transactions on Signal Processing*, vol. 58, no. 10, pp. 5369–5384, Oct 2010.

13. M. Wu, Y. Sun, S. Gupta, and J. R. Cavallaro, "Implementation of a high throughput soft mimo detector on gpu"," *Journal of Signal Processing Systems*, vol. 64, no. 1, pp. 123–136, Jul 2011. [Online]. Available: https://doi.org/10.1007/s11265-010-0523-4.

14. K. Li, M. Wu, G. Wang, and J. R. Cavallaro, "A high performance gpu-based software-defined basestation," in *48th Asilomar Conference on Signals, Systems and Computers*, Nov 2014, pp. 2060–2064.

15. M. S. Islam, C. H. Kim, and J. M. Kim, "Computationally efficient implementation of a hamming code decoder using graphics processing unit," *Journal of Communications and Networks*, vol. 17, no. 2, pp. 198–202, April 2015.

16. R. Li, Y. Dou, Y. Li, and S. Wang, "A fully parallel truncated viterbi decoder for software defined radio on gpus," in *IEEE Wireless Communications and Networking Conference (WCNC)*, April 2013, pp. 4305–4310.

17. C. S. Lin, W. L. Liu, W. T. Yeh, L. W. Chang, W. M. W. Hwu, S. J. Chen, and P. A. Hsiung, "A tiling-scheme viterbi decoder in software defined radio for gpus," in *7th International Conference on Wireless Communications, Networking and Mobile Computing*, Sept 2011, pp. 1–4.

18. F. J. Martínez-Zaldvar, A. M. Vidal-Maciá, A. Gonzalez, and V. Almenar, "Tridimensional block multiword ldpc decoding on gpus," *The Journal of Supercomputing*, vol. 58, no. 3, pp. 314–322, Dec 2011. [Online]. Available: https://doi.org/10.1007/s11227-011-0587-3.

19. R. Li, J. Zhou, Y. Dou, S. Guo, D. Zou, and S. Wang, "A multi-standard efficient column-layered ldpc decoder for software defined radio on gpus," in *IEEE 14th Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, June 2013, pp. 724–728.

20. Y. Zhao and F. C. M. Lau, "Implementation of decoders for ldpc block codes and ldpc convolutional codes based on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 663–672, March 2014.

21. J. Mitola, "Cognitive radio for flexible mobile multimedia communications," in *IEEE International Workshop on Mobile Multimedia Communications*, 1999, pp. 3–10.

22. B. McKay and K. McKay. "The Tao of Boyd: How to Master the OODA Loop". [Online]. Available: http://www.artofmanliness.com/2014/09/15/ooda-loop/.

23. C. K. Huynh and W. C. Lee, "Two-dimensional genetic algorithm for ofdm-based cognitive radio systems," in *IEEE 3rd International Conference on Communication Software and Networks*, May 2011, pp. 100–105.

24. C. J. Rieser, T. W. Rondeau, C. W. Bostian, and T. M. Gallagher, "Cognitive radio testbed: further details and testing of a distributed genetic algorithm based cognitive engine for programmable radios," in *IEEE MILCOM*, vol. 3, Oct 2004, pp. 1437–1443 Vol. 3.

25. A. Kliks, D. Triantafyllopoulou, L. D. Nardis, O. Holland, L. Gavrilovska, and A. Bantouna, "Cross-layer analysis in cognitive radio - context identification and decision making aspects," *IEEE Transactions on Cognitive Communications and Networking*, vol. 1, no. 4, pp. 450–463, Dec 2015.

26. Y. E. Morabit, F. Mrabti, and E. H. Abarkan, "Spectrum allocation using genetic algorithm in cognitive radio networks," in *Third International Workshop on RFID And Adaptive Wireless Sensor Networks (RAWSN)*, May 2015, pp. 90–93.

27. B. Bojovic, N. Baldo, and P. Dini, "A neural network based cognitive engine for ieee 802.11 wlan access point selection," in *IEEE Consumer Communications and Networking Conference (CCNC)*, Jan 2012, pp. 864–868.

28. S. Pattanayak, M. Ojha, P. Venkateswaran, and R. Nandi, "Spectrum hole detection in tv band using ann model for opportunistic radio communication," in *IEEE INDICON*, Dec 2014, pp. 1–6.

29. Y. Wu, F. Hu, Y. Zhu, and S. Kumar, "Optimal spectrum handoff control for crn based on hybrid priority queuing and multi-teacher apprentice learning," *IEEE Transactions on Vehicular Technology*, vol. 66, no. 3, pp. 2630–2642, March 2017.

30. I. AlQerm and B. Shihada, "Hybrid cognitive engine for radio systems adaptation," in *14th IEEE Annual Consumer Communications Networking Conference (CCNC)*, Jan 2017, pp. 778–783.

31. Z. j. Zhao and H. chao Lai, "A cognitive engine based on case-based reasoning quantum genetic algorithm," in *IEEE 14th International Conference on Communication Technology*, Nov 2012, pp. 224–228.

32. A. He, K. K. Bae, T. R. Newman, J. Gaeddert, K. Kim, R. Menon, L. Morales-Tirado, J. . Neel, Y. Zhao, J. H. Reed, and W. H. Tranter, "A survey of artificial intelligence for cognitive radios," *IEEE Transactions on Vehicular Technology*, vol. 59, no. 4, pp. 1578–1592, May 2010.

33. M. Ahmed, S. Hailes, V. Kolar, M. Petrova, and P. Mahonen, "A component-based architecture for cognitive radio resource management," in *4th International Conference on Cognitive Radio Oriented Wireless Networks and Communications*, June 2009, pp. 1–6.

34. X. Jing and D. Raychaudhuri, "Global control plane architecture for cognitive radio networks," in *IEEE International Conference on Communications*, June 2007, pp. 6466–6470.

35. P. D. Sutton, J. Lotze, H. Lahlou, S. A. Fahmy, K. E. Nolan, B. Ozgul, T. W. Rondeau, J. Noguera, and L. E. Doyle, "Iris: an architecture for cognitive radio networking testbeds," *IEEE Communications Magazine*, vol. 48, no. 9, pp. 114–122, Sept 2010.

36. P. D. Sutton, J. Lotze, H. Lahlou, B. zgl, S. A. Fahmy, K. E. Nolan, J. Noguera, and L. E. Doyle, "Multi-platform demonstrations using the iris architecture for cognitive radio network testbeds," in *Proceedings of the Fifth International Conference on Cognitive Radio Oriented Wireless Networks and Communications*, June 2010, pp. 1–5.

37. S. N. Khan, M. A. Kalil, and A. Mitschele-Thiel, "Distributed resource map: A database-driven network support architecture for cognitive radio ad hoc networks," in *IV International Congress on Ultra Modern Telecommunications and Control Systems*, Oct 2012, pp. 188–194.

38. J. Kleinberg and É. Tardos, *Algorithm Design.* Boston, MA: Addison-Wesley, 2006.

39. K. H. Rosen, *Discrete Mathematics and Its Applications.* New York, NY: McGraw-Hill, 2012.

40. D. S. Hochbaum, *Approximation Algorithms for NP-Hard Problems.* Boston, MA: PWS Publishing Company, 1997.

41. K. A. De Jong and W. M. Spears, "An analysis of the interacting roles of population size and crossover in genetic algorithms," in *Parallel Problem Solving from Nature*, H.-P. Schwefel and R. Männer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 38–47.

42. E.-G. Talbi, *Metaheuristics: From Design to Implementation.* Hoboken, NJ: John Wiley and Sons, 2009.

43. B. Sklar, *Digital Communications Fundamentals and Applications.* Upper Saddle River, NJ: Prentice-Hall, 2001.

44. Z. Haider, R. Hussain, I. L. Khan, A. Shakeel, B. Ijaz, and S. A. Malik, "Evaluation of capabilities of open source cognitive radio network simulators," in *13th International Wireless Communications and Mobile Computing Conference (IWCMC)*, June 2017, pp. 1814–1817.

45. V. L. Nir and B. Scheers, "Evaluation of open-source software frameworks for high fidelity simulation of cognitive radio networks," in *International Conference on Military Communications and Information Systems (ICMCIS)*, May 2015, pp. 1–6.

46. E. Weingartner, H. vom Lehn, and K. Wehrle, "A performance comparison of recent network simulators," in *IEEE International Conference on Communications*, June 2009, pp. 1–5.

47. INET Framework for OMNeT++/OMNEST. [Online]. Available: https://omnetpp.org/doc/inet/api-current/neddoc/index.html.

48. O. Helgason and S. T. Kouyoumdjieva, "Enabling multiple controllable radios in omnet++ nodes," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools '11. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011, pp. 398–401. [Online]. Available: http://dl.acm.org.afit.idm.oclc.org/citation.cfm?id=2151054.2151124.

49. S. N. Khan, M. A. Kalil, and A. Mitschele-Thiel, "crsimulator: A discrete simulation model for cognitive radio ad hoc networks in omnet++," in *6th Joint IFIP Wireless and Mobile Networking Conference (WMNC)*, April 2013, pp. 1–7.

50. N. M. Noor, N. M. Din, E. Ahmed, and A. N. A. Kadir, "Omnet++ based cognitive radio simulation network," in *7th IEEE Control and System Graduate Research Colloquium (ICSGRC)*, Aug 2016, pp. 28–33.

51. Cornet Home. [Online]. Available: https://cornet.wireless.vt.edu/.

52. T. R. Newman and J. B. Evans, "Parameter sensitivity in cognitive radio adaptation engines," in *3rd IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks*, Oct 2008, pp. 1–5.

53. T. R. Newman, "Multiple Objective Fitness Functions for Cognitive Radio Adaptation," Ph.D. dissertation, University of Kansas, 2008.

54. J. S. Milton and J. C. Arnold, *Introduction to Probability and Statistics*. Boston, MA: McGraw-Hill, 2003.

55. B. Le, "Building a Cognitive Radio: From Architecture Definition to Prototype Implementation," Ph.D. dissertation, Virginia Polytechnic Institute and State University, 2007.

56. B. C. Levy. "Table of Q Function". [Online]. Available: www.ece.ucdavis.edu/~levy/eec161/qfunc.pdf.

# Appendix A. Static Parameters for CORNET Experiments

**Table 20. General CORNET experiment parameters, not associated with either node individually and constant across all experimental runs**

| Static Variable | Value | Description |
|---|---|---|
| num_nodes | 2 | Number of nodes in scenario |
| run_time | 60 | How long scenario should be run (in seconds) |

**Table 21. Node 1 parameters which remain constant across all test cases for CORNET experiments**

| Node 1 Parameters | | |
|---|---|---|
| General | | |
| type | "CR | Node type (cognitive radio or interferer) |
| cr_type | "ecr" | Cognitive radio type (ECR or external) |
| CORNET_IP | "192.168.1.38" | IP address within CORNET |
| Network | | |
| CRTS_IP | "10.0.0.2" | |
| TARGET_IP | "10.0.0.3" | IP address of node with which this node will initially communicate |
| net_traffic_type | "poisson" | Network traffic type (stream, burst or Poisson) |
| net_mean_throughput | 2.00E+06 | Network traffic rate |
| | | |
| Cognitive engine | | |
| CE | "CE_Template" | Name of cognitive engine to be used in the scenario |
| ce_timeout_ms | 200 | How long cognitive engine should wait if node becomes unresponsive |
| | | |
| Log/report | | |
| print_metrics | 1 | Print metrics while scenario running |
| log_phy_rx | 1 | Log physical receiver metrics |
| log_phy_tx | 1 | Log physical transmitter metrics |
| log_net_rx | 1 | Log network receiver metrics |
| log_net_tx | 1 | Log network transmitter metrics |
| generate_octave_logs | 1 | Generate Octave logs for MATLAB ingest |
| | | |
| USRP | | |
| rx_freq | 8.63E+08 | Receiver frequency (Hz) |
| rx_rate | 2.00E+06 | Receiver rate |
| rx_gain | 10 | Receiver gain |
| tx_freq | 8.58E+08 | Transmitter frequency (Hz) |
| tx_rate | 2.00E+06 | Transmitter rate |
| tx_gain | 10 | Transmitter gain |
| | | |
| Liquid OFDM | | |
| duplex | "FDD" | |
| tx_gain_soft | -12 | Transmitter gain |
| tx_crc | "crc32" | Transmitter cyclic redundancy check |
| tx_fec0 | "v27" | Transmitter forward error correction scheme |
| tx_fec1 | "none" | Transmitter forward error correction scheme |
| tx_cp_len | 16 | |
| rx_cp_len | 16 | |
| | | |
| tx_subcarriers | 32 | Number of transmitter subcarriers |
| tx_subcarrier_alloc_method | "standard" | Transmitter subcarrier allocation method |
| tx_guard_subcarriers | 4 | Number of transmitter guard subcarriers |
| tx_central_nulls | 6 | Number of transmitter central nulls |
| tx_pilot_freq | 4 | Transmitter pilot frequency |
| | | |
| rx_subcarriers | 32 | Number of receiver subcarriers |
| rx_subcarrier_alloc_method | "standard" | Receiver subcarrier allocation method |
| rx_guard_subcarriers | 4 | Number of receiver guard subcarriers |
| rx_central_nulls | 6 | Number of receiver central nulls |
| rx_pilot_freq | 4 | Receiver pilot frequency |

**Table 22. Node 2 parameters which remain constant across all test cases for CORNET experiments**

| Node 1 Parameters | | |
|---|---|---|
| General | | |
| type | "CR | Node type (cognitive radio or interferer) |
| cr_type | "ecr" | Cognitive radio type (ECR or external) |
| CORNET_IP | "192.168.1.37" | IP address within CORNET |
| Network | | |
| CRTS_IP | "10.0.0.3" | |
| TARGET_IP | "10.0.0.2" | IP address of node with which this node will initially communicate |
| net_traffic_type | "poisson" | Network traffic type (stream, burst or Poisson) |
| net_mean_throughput | 2.00E+06 | Network traffic rate |
| | | |
| Cognitive engine | | |
| CE | "CE_Template" | Name of cognitive engine to be used in the scenario |
| ce_timeout_ms | 200 | How long cognitive engine should wait if node becomes unresponsive |
| | | |
| Log/report | | |
| print_metrics | 1 | Print metrics while scenario running |
| log_phy_rx | 1 | Log physical receiver metrics |
| log_phy_tx | 1 | Log physical transmitter metrics |
| log_net_rx | 1 | Log network receiver metrics |
| log_net_tx | 1 | Log network transmitter metrics |
| generate_octave_logs | 1 | Generate Octave logs for MATLAB ingest |
| | | |
| USRP | | |
| rx_freq | 8.63E+08 | Receiver frequency (Hz) |
| rx_rate | 2.00E+06 | Receiver rate |
| rx_gain | 10 | Receiver gain |
| tx_freq | 8.58E+08 | Transmitter frequency (Hz) |
| tx_rate | 2.00E+06 | Transmitter rate |
| tx_gain | 10 | Transmitter gain |
| | | |
| Liquid OFDM | | |
| duplex | "FDD" | |
| tx_gain_soft | -12 | Transmitter gain |
| tx_crc | "crc32" | Transmitter cyclic redundancy check |
| tx_fec0 | "v27" | Transmitter forward error correction scheme |
| tx_fec1 | "none" | Transmitter forward error correction scheme |
| tx_cp_len | 16 | |
| rx_cp_len | 16 | |
| | | |
| tx_subcarriers | 32 | Number of transmitter subcarriers |
| tx_subcarrier_alloc_method | "standard" | Transmitter subcarrier allocation method |
| tx_guard_subcarriers | 4 | Number of transmitter guard subcarriers |
| tx_central_nulls | 6 | Number of transmitter central nulls |
| tx_pilot_freq | 4 | Transmitter pilot frequency |
| | | |
| rx_subcarriers | 32 | Number of receiver subcarriers |
| rx_subcarrier_alloc_method | "standard" | Receiver subcarrier allocation method |
| rx_guard_subcarriers | 4 | Number of receiver guard subcarriers |
| rx_central_nulls | 6 | Number of receiver central nulls |
| rx_pilot_freq | 4 | Receiver pilot frequency |

# Appendix B. Code Listings for OMNeT++ Cognitive Engine

**Listing B.1. Listing for OMNeT++ cognitive engine implementation.**

```
/*
 * CRCENetworkEngine.cpp
 *
 * Created on: Jan 13, 2018
 * Author: Dan Hart
 */

#include "CRCENetworkEngine.h"

Define_Module(CRCENetworkEngine);

CRCENetworkEngine::CRCENetworkEngine(){
}

CRCENetworkEngine::~CRCENetworkEngine() {
}

double CRCENetworkEngine::calculatePSKBER(double modulationOrder, double bitrate,
        double bandwidth, double power, double noise, double r) {

        /*
         * Evaluate the argument to the Q function and then look up the appropriate
         * value in the Q function table
         */
        double powerWatts = pow(10, (power - 30) / 10);
        double noiseWatts = pow(10, (noise - 30) / 10);
        double qFunctionArgument;
        if(noise == 0) {
                qFunctionArgument = 9;
        }
        else {
                qFunctionArgument = sqrt((1 / double (bitrate / modulationOrder)) *
                bandwidth) * sin(M_PI / pow(2, modulationOrder)) * sqrt((r * 2 *
                powerWatts) / noiseWatts);
        }
        double qFunctionResult = this->evaluateQFunction(qFunctionArgument);

        //Calculate the BER using intermediate results and return
        return (2 / modulationOrder) * qFunctionResult;
}

double CRCENetworkEngine::calculateQAMBER(double modulationOrder, double bitrate,
        double bandwidth, double power, double noise, double r) {
        /*
         * Evaluate the argument to the Q function and then look up the appropriate
         * value in the Q function table
         */
        double powerWatts = pow(10, (power - 30) / 10);
        double noiseWatts = pow(10, (noise - 30) / 10);
        double qFunctionArgument;
        if(noise == 0) {
                qFunctionArgument = 9;
        }
        else {
                qFunctionArgument = sqrt((3 * r *  (1 / double (bitrate /
                modulationOrder)) * bandwidth * powerWatts) /
                ((modulationOrder - 1) * noiseWatts));
        }
        double qFunctionResult = this->evaluateQFunction(qFunctionArgument);
```

```
        //Calculate the BER using intermediate results and return
        return (4 / modulationOrder) * ((sqrt(pow(2, modulationOrder)) - 1) /
        sqrt(pow(2, modulationOrder))) * qFunctionResult;
}

//Method to compute fitness score; uses other helper methods
double CRCENetworkEngine::computeFitnessScore(std::string parameterEncoding) {
        /*
         * Radio parameter encoding:
         * First bit: modulation scheme
         * Next 4 bits encode modulation order
         * Next 5 bits encode power
         * Final 5 bits encode bandwidth
         */

        //Extract the encoded modulation scheme from the string
        std::string modulationScheme = parameterEncoding.substr(0, 1);

        //Extract the encoded modulation order from the string
        std::string modulationOrderString = parameterEncoding.substr(1, 4);

        //Extract the encoded power setting from the string
        std::string powerString = parameterEncoding.substr(5, 5);

        //Extract the encoded bandwidth setting from the string
        std::string bandwidthString = parameterEncoding.substr(10, 5);

        //Convert the encoded parameter values

        /*
         * Convert modulation order, power and bandwidth;
         * ADJUST POWER AND BANDWIDTH IN ACCORDANCE WITH REQUIRED OFFSETS TO
         * ENSURE THEY FALL WITHIN THE SPECIFIED RANGE!!
         */
        double modulationOrder = convertStringToDouble(modulationOrderString);
        double power = convertStringToDouble(powerString) + 4;
        double bandwidth = this->convertStringToDouble(bandwidthString) + 1;

        double r = 1;

        double bitErrorRate;

        if(modulationScheme.compare("0") == 0) {
                //Using PSK modulation
                //Convert bandwidth from MHz (current) to Hz (required)
                bitErrorRate = this->calculatePSKBER(modulationOrder, this->bitrate,
                bandwidth * 1000000, power, this->receiverNoiseLevel, r);
        }
        else {
                //Using QAM
                //Convert bandwidth from MHz (current) to Hz (required)
                bitErrorRate = this->calculateQAMBER(modulationOrder, this->bitrate,
                bandwidth * 1000000, power, this->receiverNoiseLevel, r);
        }

        double latencyFitness = this->computeLatencyFitness(bitErrorRate);

        double powerFitness = this->computePowerFitness(modulationOrder, power,
        bandwidth);

        double throughputFitness = this->computeThroughputFitness(bitErrorRate);

        return this->latencyWeight * latencyFitness + this->powerWeight *
        powerFitness + this->throughputWeight * throughputFitness;
}
```

```
double CRCENetworkEngine::computeLatencyFitness(double bitErrorRate) {
        double errorlessLatency = ceil((this->packetSize + this->packetHeaderSize +
        (((this->packetSize + this->packetHeaderSize) / this->datagramSize) *
        this->datagramHeaderSize)) / this->frameSize) * this->frameLatency;

        double errorPenalty = 1 / (1 - (1 - (pow((1 - bitErrorRate),
        (this->frameSize + this->frameHeaderSize) * 8))));

        return errorlessLatency / (errorPenalty * errorlessLatency);
}

double CRCENetworkEngine::computePowerFitness(double modulationOrder, double power,
double bandwidth) {
        /*
         * Calculate the three components individually to try and keep things a
         * little more readable/manageable
         */
        double componentOne = 1 - (this->alpha * (((this->maxPower +
        this->maxBandwidth) - (power + bandwidth)) / (this->maxPower +
        this->maxBandwidth)));

        double componentTwo = this->beta * ((log2(this->maxModulationOrder) -
        log2(modulationOrder)) / log2(this->maxModulationOrder));

        double componentThree = this->lambda * (((this->bitrate /
        this->maxModulationOrder) - (this->bitrate / modulationOrder)) /
        (this->bitrate / this->maxModulationOrder));

        //Return the sum of the three components
        return componentOne + componentTwo + componentThree;
}

double CRCENetworkEngine::computeThroughputFitness(double bitErrorRate) {
        return (this->frameSize / (this->frameSize + this->frameHeaderSize +
        this->packetHeaderSize + this->datagramHeaderSize)) * pow((1 - bitErrorRate),
        this->frameSize + this->frameHeaderSize) * this->codingRate * this->TDD;
}

double CRCENetworkEngine::convertStringToDouble(std::string binaryValue) {
        double doubleValue = 0;
        for(int index = binaryValue.size() - 1; index >= 0; index--) {
                if(!(binaryValue.substr(index, 1).compare("1"))) {
                        doubleValue += pow(2, binaryValue.size() - index - 1);
                }
        }
        return doubleValue;
}

//Method to evaluate Q function via table lookup
double CRCENetworkEngine::evaluateQFunction(double qFunctionArgument) {
        int index = 0;

        //Explicitly handle case where argument is 0; breaks code otherwise
        if(qFunctionArgument == 0) {
                return .05;
        }

        //Find the second row to be used in interpolating the correct Q function value
        while(this->qFunctionTable[index][0] < qFunctionArgument && index < 180) {
                index++;
        }

        if(index == 0) {
                //The Q function argument passed in is less than 0, an error; break
                std::exit(1);
```

```
        }
        //Interpolate the Q function value and return
        double index1 = this->qFunctionTable[index - 1][0];
        double index2 = this->qFunctionTable[index][0];
        double value1 = this->qFunctionTable[index - 1][1];
        double value2 = this->qFunctionTable[index][1];

        double qFunctionValue = (value1 + ((value2 - value1) / (index2 - index1)) *
        (qFunctionArgument - index1));

        /*
         * Code to test evaluateQFunction method
         * std::cout << qFunctionValue << "\n";
         */

        return qFunctionValue;
}

/*
 * Method to generate a random bandwidth encoding; will be used in initializing
 * the population
 */
std::string CRCENetworkEngine::generateBandwidthString() {
        //5 total bits

        //Bandwidth gene which will be iteratively built
        std::string bandwidthGene = "";

        /*
         * For each bit position, generate a random number; if it is even, add a 0,
         * otherwise add a 1
         */
        for(int index = 0; index < 5; index++) {
                //For each bit, generate a random number and test to assign bit
                if(rand() % 2 == 0) {
                        bandwidthGene += "0";
                        }
                else {
                        bandwidthGene += "1";
                }
        }
        return bandwidthGene;
}

/*
 * Method to generate a random modulation scheme encoding; will be used in
 * initializing the population
 */
std::string CRCENetworkEngine::generateModulationScheme() {
        if(rand() % 2 == 0) {
                return "0";
        }
        else {
                return "1";
        }
}

/*
 * Method to generate a random modulation order encoding; will be used in
 * initializing the population
 */
std::string CRCENetworkEngine::generateModulationOrder(bool usingQAM) {
        /*
         * Boolean values to track if first bit is a one (all others must be 0)
         * and if, when we get to the last bit, all bits are 0 (last bit must be a 1)
         */
```

```
        bool firstOne = false;
        bool allZeros = true;

        std::string modulationOrderString;

        for(int index = 0; index < 4; index++) {
                if(index == 0) {
                        if(rand() % 8 == 0) {
                                /*
                                 * Using 8 here to cut down one the number of times
                                 * the first bit is set to one as that sets all
                                 * other bits
                                 */
                                modulationOrderString += "1";
                                firstOne = true;
                                allZeros = false;
                        }
                        else {
                                modulationOrderString += "0";
                        }
                }
                else if(index > 0 && firstOne) {
                        //The first bit is a 1, assign a 0
                        modulationOrderString += "0";
                }
                else if(index == 3 && (allZeros || usingQAM)) {
                        if(usingQAM) {
                                if(allZeros) {
                                        /*
                                         * Need to set preceding bit (at position 2)
                                         * to 1 and set this bit to 0
                                         */
                                        modulationOrderString.replace(2, 1, "1");
                                }
                                modulationOrderString += "0";
                        }
                        else {
                                if(allZeros) {
                                /*
                                 * Looking at the last bit and all preceding bits
                                 * are 0, must assign a 1
                                 */
                                modulationOrderString += "1";
                                }
                        }
                }
                else {
                        /*
                         * Generate random number; if it is even, assign a 0,
                         * otherwise assign a 1
                         */
                        if(rand() % 2 == 0) {
                                modulationOrderString += "0";
                        }
                        else {
                                modulationOrderString += "1";
                                allZeros = false;
                        }
                }
        }
        return modulationOrderString;
}

/*
 * Method to generate a random modulation power encoding;
 * will be used in initializing the population
```

```
 */
std::string CRCENetworkEngine::generatePowerString() {
        //5 total bits

        //Bandwidth gene which will be iteratively built
        std::string powerGene = "";

        /*
         * For each bit position, generate a random number;
         * if it is even, add a 0, otherwise add a 1
         */
        for(int index = 0; index < 5; index++) {
                //For each bit, generate a random number and test to assign bit
                if(rand() % 2 == 0) {
                        powerGene += "0";
                        }
                else {
                        powerGene += "1";
                }
        }
        return powerGene;
}

//Simple getter method to return the population size
double CRCENetworkEngine::getPopulationSize() {
        return this->populationSize;
}

/*
 * Method to handle self-message, run the genetic algorithm and re-schedule
 * a new message for 5 seconds in the future
 */
void CRCENetworkEngine::handleMessage(omnetpp::cMessage* incomingMessage) {
        if(incomingMessage->getKind() == this->run_genetic_algorithm) {
                //Run the genetic algorithm
                std::string newParameterEncoding = this->runGeneticAlgorithm();

                /*
                 * Call each radio's parameter update method
                 * to update operating parameters
                 */
                for(int index = 0; index < par("numRadioNodes").longValue(); index++) {
                        //Form radio node's name
                        std::string radioName = "cr_node_" + std::to_string(index);

                        //Get reference to node
                        CRCENetworkRadio* radio =
                        (CRCENetworkRadio*) this->getParentModule()->
                        getSubmodule(radioName.c_str())->
                        getModuleByPath(".wlan[0]")->getSubmodule("radio");

                        //Call the radio's parameter update function
                        radio->updateParameters(newParameterEncoding);
                }
        }

        //Save chromosome to file
        std::fstream chromosomesFile;
        chromosomesFile.open("Chromosomes.txt", std::fstream::app);

        chromosomesFile << this->geneticAlgorithmRunNumber << "\t\t" <<
        this->geneticAlgorithmPopulation->getPopulationMember(0) << "\t\t" <<
        SIMTIME_DBL(simTime()) << std::endl;

        chromosomesFile.close();
```

```
        this->geneticAlgorithmRunNumber = this->geneticAlgorithmRunNumber + 1;

        if(this->geneticAlgorithmRunNumber % 40 == 0) {
                //Run the algorithm 40 times at each power level
                this->geneticAlgorithmRunNumber = 0;

                //Update background noise level
                IsotropicScalarBackgroundNoise* backgroundNoise =
                (IsotropicScalarBackgroundNoise*) this->getParentModule()->
                getSubmodule("networkRadioMedium")->getSubmodule("backgroundNoise");

                double noiseInWatts = backgroundNoise->getPower().get();

                double noiseInDBm = 10 * log10(1000 * noiseInWatts) + 10;
                backgroundNoise->setBackgroundNoisePower(noiseInDBm);

                this->receiverNoiseLevel = noiseInDBm;
        }

        //Schedule a new message to run genetic algorithm 5 seconds in the future
        omnetpp::cMessage* runAlgorithmMessage = new omnetpp::cMessage();
        runAlgorithmMessage->setKind(this->run_genetic_algorithm);
        this->scheduleAt(omnetpp::simTime() + 5, runAlgorithmMessage);
}


/*
 * Method to initialize engine; creates the engine,
 * sets parameters appropriately and schedules first event
 */
void CRCENetworkEngine::initialize(int initializationStage) {
        /*
         * Initialize the population size, weights,
         * maximum parameter settings and the population itself
         */
        this->alpha = par("alpha");
        this->beta = par("beta");
        this->bitrate = par("bitrate");
        this->chromosomeLength = par("chromosomeLength");
        this->codingRate = par("codingRate");
        this->datagramHeaderSize = par("datagramHeaderSize");
        this->datagramSize = par("datagramSize");
        this->frameHeaderSize = par("frameHeaderSize");
        this->frameLatency = par("frameLatency");
        this->frameSize = par("frameSize");
        this->geneticAlgorithmRunNumber = 1;
        this->lambda = par("lambda");
        this->latencyWeight = par("latencyWeight");
        this->maxBandwidth = par("maxBandwidth");
        this->maxModulationOrder = par("maxModulationOrder");
        this->maxPower = par("maxPower");
        this->numberOfGenerations = par("numberOfGenerations");
        this->packetHeaderSize = par("packetHeaderSize");
        this->packetSize = par("packetSize");
        this->populationSize = par("populationSize");
        this->powerWeight = par("powerWeight");
        this->receiverNoiseLevel = -20;
        this->throughputWeight = par("throughputWeight");
        this->TDD = par("TDD");

        //Store the message kind that will be used to tell the engine to run
        this->run_genetic_algorithm = 253;

        //Create the engine's population
        this->geneticAlgorithmPopulation = new CRCEPopulation(this->populationSize);
```

```
        //Add header to chromosomes file
        std::remove("Chromosomes.txt");
        std::ofstream chromosomesFile("Chromosomes.txt");

        chromosomesFile << "Run Number" << "\t" << "Chromosome" << "\t\t" <<
        "Simulation Time" << std::endl;

        chromosomesFile.close();

        //Initialize the Q function table
        //Read in the values for the Q function table
        std::ifstream qFunctionTableText("QFunctionTable.txt");
        std::string nextLine;
        int index = 0;
        double value1 = 0;
        double value2 = 0;

        while(std::getline(qFunctionTableText, nextLine, '\n')) {
                std::istringstream qFunctionTableTextStream(nextLine);
                qFunctionTableTextStream >> value1;
                qFunctionTableTextStream >> value2;

                this->qFunctionTable[index][0] = value1;
                this->qFunctionTable[index][1] = value2;

                index++;
        }

        //Iitialize the population
        this->initializePopulation();

        /*
         * Create a new message,
         * set its kind to "run_genetic_algorithm"
         * and schedule it for 5 seconds in the future
         */
        EV << "Scheduling first message to tell engine to run";
        omnetpp::cMessage* runAlgorithmMessage = new omnetpp::cMessage();
        runAlgorithmMessage->setKind(this->run_genetic_algorithm);
        this->scheduleAt(omnetpp::simTime() + 5, runAlgorithmMessage);
}

//Initializes the population with randomly generated chromosomes
void CRCENetworkEngine::initializePopulation() {
        //Initialize random number generator to random value
        srand(time(NULL));

        int index;

        for(index = 0; index < this->populationSize; index++) {
                /*
                 * Randomly generate a new parameter encoding,
                 * evaluate its fitness and add to population
                 */
                std::string modulationSchemeString =
                this->generateModulationScheme();

                std::string modulationOrderString =
                this->generateModulationOrder(
                modulationSchemeString.compare("1") == 0);

                std::string powerString = this->generatePowerString();

                std::string bandwidthString = this->generateBandwidthString();

                if(modulationSchemeString.compare("1") == 0 &&
```

```
                modulationOrderString.substr(3,1).compare("1") == 0) {
                    std::cout << "ERROR!" << std::endl;
                }

                std::string chromosome = modulationSchemeString +
                modulationOrderString + powerString + bandwidthString;

                double fitnessValue = this->computeFitnessScore(chromosome);

                this->geneticAlgorithmPopulation->
                addMemberToPopulation(chromosome, fitnessValue, true);
        }
}

std::string CRCENetworkEngine::mutateChromosome(std::string recombinedChromosome) {
        /*
         * Extract the four parts of the chromosome;
         * the modulation order will be mutated differently than the other parts
         * Radio parameter encoding:
         * First bit: modulation scheme
         * Next 4 bits encode modulation order
         * Next 5 bits encode power
         * Final 5 bits encode bandwidth
         */

        //Extract the encoded modulation scheme from the string
        std::string modulationScheme = recombinedChromosome.substr(0, 1);

        //Extract the encoded modulation order from the string
        std::string modulationOrderString = recombinedChromosome.substr(1, 4);

        //Extract the encoded power setting from the string
        std::string powerString = recombinedChromosome.substr(5, 5);

        //Extract the encoded bandwidth setting from the string
        std::string bandwidthString = recombinedChromosome.substr(10, 5);

        std::string mutatedChromosome;

        //Mutate modulation scheme
        if(rand() % (int) this->chromosomeLength == 0) {
                if(modulationScheme.compare("0") == 0) {
                        mutatedChromosome += "1";
                }
                else {
                        mutatedChromosome += "0";
                }
        }
        else {
                mutatedChromosome += modulationScheme;
        }

        /*
         * Mutate modulation order
         * If first bit is made a 1, make all other bits 0
         * When mutating last bit,
         * if it is changed to a zero, must make sure at least one other bit is a 1
         */

        //Boolean to determine if first bit is a 1
        bool firstBitOne = false;
        //Boolean to determine if all bits but the last are zeros
        bool allBitsZero = true;

        //Boolean to determine if we are using QAM
        bool usingQAM = (mutatedChromosome.substr(0,1).compare("1") == 0);
```

```
//Iterate through each bit
for(int index = 0; index < modulationOrderString.size() ; index++) {
        /*
         * For the first bit, if we should mutate and it is a 1,
         * make it a zero; if it is a 0, make it a one
         * If we make the first bit a 1, set firstBitOne to true
         * to make sure all other bits are set to 0
         * (required for a valid encoding)
         */
        if(index == 0) {
                if(rand() % (int) this->chromosomeLength == 0) {
                        if(modulationOrderString.substr(0,1).
                        compare("0") == 0) {

                                //Had a zero, make it a 1
                                mutatedChromosome += "1";
                                firstBitOne = true;
                                allBitsZero = false;
                        }
                        else {
                                //Had a 1, make it a zero
                                mutatedChromosome += "0";
                        }
                }
                else {
                        mutatedChromosome +=
                        modulationOrderString.substr(index, 1);

                        if(modulationOrderString.substr(index, 1).
                        compare("1") == 0) {

                                firstBitOne = true;
                                allBitsZero = false;
                        }
                }
        }

        //If the first bit is a one, set all other bits to zero
        else if(firstBitOne && index != 0) {
                mutatedChromosome += "0";
        }

        //Looking at the last bit
        else if(index == 3) {
                //If all preceding bits are zero, must set this bit to 1
            if(usingQAM) {
                if(allBitsZero) {
                    //Replace preceding bit with a 1 and set this one to 0
                    mutatedChromosome.replace(3, 1, "1");
                }
                mutatedChromosome += "0";
            }
            else if(allBitsZero) {
                        mutatedChromosome += "1";
                }
                //At least two other bits are 1, mutate this bit
                else {
                        //Mutate bit
                        if(rand() % (int) this->chromosomeLength == 0) {
                                //Had a zero, make it a one
                                if(modulationOrderString.substr(index, 1).
                                compare("0") == 0) {

                                        mutatedChromosome += "1";
```

```
                                }

                                //Had a one, make it a zero
                                else {
                                        mutatedChromosome += "0";
                                }
                        }
                        //Don't mutate bit, just add original bit
                        else {
                                mutatedChromosome +=
                                modulationOrderString.substr(index, 1);
                        }
                }
        }

        /*
         * For the second 2 bits, if the first bit is not a one,
         * probabilistically mutate the bits
         */
        else {
                if(rand() % (int) this->chromosomeLength == 0) {
                        //Mutate bit
                        if(modulationOrderString.substr(index, 1).
                        compare("0") == 0) {

                                /*
                                 * If bit was zero,
                                 * make it one, set allBitsZero false
                                 */
                                mutatedChromosome += "1";
                                allBitsZero = false;
                        }
                        else {
                                //Otherwise, bit was a one so add a zero
                                mutatedChromosome += "0";
                        }
                }
                else {
                        /*
                         * Do not mutate bit; add original bit,
                         * setting allBitsZero to false if we add a one
                         */
                        mutatedChromosome += modulationOrderString.
                        substr(index, 1);

                        if(modulationOrderString.substr(index, 1).
                        compare("1") == 0) {

                                allBitsZero = false;
                        }
                }
        }
}

//Need to halt if using QAM and odd modulation order
bool ifUsingQAM = mutatedChromosome.substr(0,1).compare("1") == 0;
bool oddModulationOrder = mutatedChromosome.substr(4,1).compare("1") == 0;

/*
 * Mutate power and bandwidth
 * Here, don't need complicated validation as with the modulation order
 * Simply probabilistically flip bits
 */

//Mutate power
for(int index = 0; index < powerString.size(); index++) {
```

```
                if(rand() % (int) this->chromosomeLength == 0) {
                        //Mutate bit
                        if(powerString.substr(index, 1).compare("0") == 0) {
                                //Had a zero, add a one
                                mutatedChromosome += "1";
                        }
                        else {
                                //Had a one, add a zero
                                mutatedChromosome += "0";
                        }
                }
                else {
                        //Don't mutate bit
                        mutatedChromosome += powerString.substr(index, 1);
                }
        }

        //Mutate bandwidth
        for(int index = 0; index < bandwidthString.size(); index++) {
                if(rand() % (int) this->chromosomeLength == 0) {
                        //Mutate bit
                        if(bandwidthString.substr(index, 1).compare("0") == 0) {
                                //Had a zero, add a one
                                mutatedChromosome += "1";
                        }
                        else {
                                //Had a one, add a zero
                                mutatedChromosome += "0";
                        }
                }
                else {
                        //Don't mutate bit
                        mutatedChromosome += bandwidthString.substr(index, 1);
                }
        }

        //Return mutated chromosome
        return mutatedChromosome;
}

/*
 * Recombines chromosomes;
 * splits chromosomes into genes and probabilistically picks one of the two genes
 */
std::string CRCENetworkEngine::recombineChromosomes(std::string firstChromosome,
std::string secondChromosome) {

        //Extract the four parts of the two chromosomes

        /*
         * Radio parameter encoding:
         * First bit: modulation scheme
         * Next 4 bits encode modulation order
         * Next 5 bits encode power
         * Final 5 bits encode bandwidth
         */

        //Extract the encoded modulation scheme from the string
        std::string firstModulationScheme = firstChromosome.substr(0, 1);
        std::string secondModulationScheme = secondChromosome.substr(0, 1);

        //Extract the encoded modulation order from the string
        std::string firstModulationOrderString = firstChromosome.substr(1, 4);
        std::string secondModulationOrderString = secondChromosome.substr(1, 4);

        //Extract the encoded power setting from the string
```

```
std::string firstPowerString = firstChromosome.substr(5, 5);
std::string secondPowerString = secondChromosome.substr(5, 5);

//Extract the encoded bandwidth setting from the string
std::string firstBandwidthString = firstChromosome.substr(10, 5);
std::string secondBandwidthString = secondChromosome.substr(10, 5);

std::string recombinedChromosome;

/*
 * Generate a random number 4 times
 * If the number is even, use the gene from the first chromosome
 * Otherwise use the gene from the second chromosome
 */

//Modulation scheme
if(rand() % 2 == 0) {
        recombinedChromosome += firstModulationScheme;
}
else {
        recombinedChromosome += secondModulationScheme;
}

//Modulation order
if(rand() % 2 == 0) {
        recombinedChromosome += firstModulationOrderString;
}
else {
        recombinedChromosome += secondModulationOrderString;
}

/*
 * Need to check if QAM paired with odd modulation order
 * if so, make the third bit a 1 and zero out the fourth bit
 */
bool ifUsingQAM = recombinedChromosome.substr(0,1).compare("1") == 0;
bool oddModulationOrder = recombinedChromosome.substr(4,1).compare("1") == 0;
if(ifUsingQAM && oddModulationOrder) {
        if(!(recombinedChromosome.substr(1,1).compare("1") == 0 ||
        recombinedChromosome.substr(2,1).compare("1") == 0 ||
        recombinedChromosome.substr(3,1).compare("1") == 0)) {

                int indexToReplace = (rand() % 3) + 1;
                recombinedChromosome.replace(indexToReplace, 1, "1");
        }
        recombinedChromosome.replace(4, 1, "0");
}

ifUsingQAM = recombinedChromosome.substr(0,1).compare("1") == 0;
oddModulationOrder = recombinedChromosome.substr(4,1).compare("1") == 0;

//Power
if(rand() % 2 == 0) {
        recombinedChromosome += firstPowerString;
}
else {
        recombinedChromosome += secondPowerString;
}

//Bandwidth
if(rand() % 2 == 0) {
        recombinedChromosome += firstBandwidthString;
}
else {
        recombinedChromosome += secondBandwidthString;
}
```

```
        //Return the recombined chromosome
        return recombinedChromosome;
}

//Runs the genetic algorithm
std::string CRCENetworkEngine::runGeneticAlgorithm() {
        for(int index = 0; index < this->numberOfGenerations; index++) {
                //Randomly choose two chromosomes from population for recombination
                std::string firstChromosome = this->geneticAlgorithmPopulation->
                getPopulationMember(rand() % ((int) this->populationSize));

                std::string secondChromosome = this->geneticAlgorithmPopulation->
                getPopulationMember(rand() % (int) this->populationSize);

                std::string recombinedChromosome =
                this->recombineChromosomes(firstChromosome, secondChromosome);

                std::string mutatedChromosome =
                this->mutateChromosome(recombinedChromosome);

                double newFitnessValue =
                this->computeFitnessScore(mutatedChromosome);

                this->geneticAlgorithmPopulation->
                addMemberToPopulation(mutatedChromosome, newFitnessValue, false);
        }

        return this->geneticAlgorithmPopulation->getPopulationMember(1);
}

//Sets the population's size, if needed
void CRCENetworkEngine::setPopulationSize(double newPopulationSize) {
        //Set the population size to the value provided
        this->populationSize = newPopulationSize;
}
```

**Listing B.2. Header file for OMNeT++ cognitive engine implementation.**

```
/*
 * CRCENetworkEngine.h
 *
 * Created on: Jan 13, 2018
 * Author: Dan Hart
 */

#ifndef CRCENETWORKENGINE_H_
#define CRCENETWORKENGINE_H_

#include <string>
#include <math.h>
#include <fstream>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>

#include "omnetpp.h"
#include "CRCEPopulation.h"
#include "CRCENetworkRadio.h"
#include "inet/physicallayer/apskradio/packetlevel/APSKScalarTransmitter.h"
#include "inet/physicallayer/backgroundnoise/IsotropicScalarBackgroundNoise.h"

class CRCENetworkEngine : public omnetpp::cSimpleModule{
        public:
        //DATA MEMBERS

        //Power weighting factor
        double alpha;
        //Power weighting factor
        double beta;
        //Bitrate; ASSUMED IN BPS BY PROGRAM!!
        double bitrate;
        //Chromosome length
        double chromosomeLength;
        //Coding rate
        double codingRate;
        //Datagram header size, in bytes
        double datagramHeaderSize;
        //Datagram size
        double datagramSize;
        //Frame header size, in bytes
        double frameHeaderSize;
        //Latency for an individual frame
        double frameLatency;
        //Frame size, in bytes
        double frameSize;
        //Population the engine will use in generating future generations
        CRCEPopulation* geneticAlgorithmPopulation;
        //Variable to track how many times the genetic algorithm has been run
        int geneticAlgorithmRunNumber;
        //Power weighting factor
        double lambda;
        //Latency weighting factor
        double latencyWeight;
        //Maximum allowable bandwidth
        double maxBandwidth;
        //Maximum allowable modulation order
        double maxModulationOrder;
        //Maximum allowable power
        double maxPower;
        //Number of generations to generate when the genetic algorithm runs
        double numberOfGenerations;
```

```
//Packet header size, in bytes
double packetHeaderSize;
//Packet size, in bytes
double packetSize;
//Size of the population
double populationSize;
//Power weighting factor
double powerWeight;
//Two-dimensional array to hold the Q function table
double qFunctionTable[181][2];
//Message kind for which to check to see if genetic algorithm should be run
short run_genetic_algorithm;
//Time-Division duplexing factor
double TDD;
//Throughput weighting factor
double throughputWeight;

//Variables to keep track of the noise power at each receiver
double receiverNoiseLevel;



//METHODS

//Method to calculate the BER when using PSK modulation
double calculatePSKBER(double modulationOrder, double bitrate,
double bandwidth, double power, double noise, double r);

//Method to calculate the BER when using QAM
double calculateQAMBER(double modulationOrder, double bitrate,
double bandwidth, double power, double noise, double r);

//Method to calculate the total fitness score value
double computeFitnessScore(std::string parameterEncoding);
//Method to compute the latency fitness value
double computeLatencyFitness(double bitErrorRate);
//Method to compute the power fitness value
double computePowerFitness(double modulationOrder, double power,
double bandwidth);

//Method to compute the throughput fitness value
double computeThroughputFitness(double bitErrorRate);
//Method to convert gene encodings to their integer values
double convertStringToDouble(std::string binaryValue);
//Method to look up/interpolate the Q function value from the table
double evaluateQFunction(double qFunctionArgument);
//Method to generate a random bandwidth encoding
std::string generateBandwidthString();
//Method to generate a random modulation scheme encoding
std::string generateModulationScheme();
//Method to generate a random modulation order encoding
std::string generateModulationOrder(bool usingQAM);
//Method to generate a random modulation power encoding
std::string generatePowerString();
//Getter method to return the population's size
double getPopulationSize();
//Used to handle the self-message and run genetic algorithm
void handleMessage(omnetpp::cMessage* incomingMessage);
/*
 * Method to initialize the engine;
 * used to create the engine,
 * set parameters appropriately and schedule first event
 */
void initialize(int initializationStage);
//Method to initialize the population
void initializePopulation();
```

```
        //Method to mutate recombined chromosome
        std::string mutateChromosome(std::string recombinedChromosome);
        //Method to perform recombination (crossover) of two chromosomes
        std::string recombineChromosomes(std::string firstChromosome,
        std::string secondChromosome);

        //Method to run the engine's genetic algorithm
        std::string runGeneticAlgorithm();
        //Setter method to set the population's size
        void setPopulationSize(double newPopulationSize);



        //Constructor
        CRCENetworkEngine();
        //Destructor
        virtual ~CRCENetworkEngine();
};

#endif /* CRCENETWORKENGINE_H_ */
```

**Listing B.3. Code listing for new radio type, extended from existing "APSKScalarRadio" module. All functionality is the same as provided in "APSKScalarRadio" except a new function to update operating parameters.**

```
/*
 * CRCENetworkRadio.cpp
 *
 * Created on: Jan 12, 2018
 * Author: Dan Hart
 */

//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program.  If not, see http://www.gnu.org/licenses/.
//

#include "CRCENetworkRadio.h"

//Define_Module(CRCENetworkRadio);

CRCENetworkRadio::CRCENetworkRadio() {
        // TODO Auto-generated constructor stub
}

CRCENetworkRadio::~CRCENetworkRadio() {
        // TODO Auto-generated destructor stub
}

int CRCENetworkRadio::convertStringToInt(std::string binaryValue) {
        int integerValue = 0;
        for(int index = binaryValue.size() - 1; index >= 0; index--) {
                if(!(binaryValue.substr(index, 1).compare("1"))) {
                        integerValue += pow(2, binaryValue.size() - index - 1);
                }
        }
        return integerValue;
}

void CRCENetworkRadio::updateParameters(std::string radioParameters) {
        /*
        * Radio parameter encoding:
        * First bit: modulation scheme
        * Next 4 bits encode modulation order
        * Next 5 bits encode power
        * Final 5 bits encode bandwidth
        */

        //Extract the encoded modulation scheme from the string
        std::string modulationScheme = radioParameters.substr(0, 1);

        //Extract the encoded modulation order from the string
        std::string modulationOrderString = radioParameters.substr(1, 4);

        //Extract the encoded power setting from the string
        std::string powerString = radioParameters.substr(5, 5);
```

```
        //Extract the encoded bandwidth setting from the string
        std::string bandwidthString = radioParameters.substr(10, 5);

        //Convert the encoded parameter values

        //Convert modulation scheme and order (0 is PSK, 1 is QAM)
        int modulationOrder = convertStringToInt(modulationOrderString);

        //bool usingQAM = modulationScheme.compare("1") == 0;
        //bool oddModulationOrder = (modulationOrder % 2 != 0);

        //Calculate constellation size
        //int constellationSize = pow(2, modulationOrder);

        std::string findModulationArgument;
        if(modulationScheme.compare("0") == 0) {
                //Using PSK modulation
                findModulationArgument = "MPSK-" + this->toString(modulationOrder);
                this->setModulation(APSKModulationBase::
                findModulation(findModulationArgument.c_str()));
        }
        else {
                //Using QAM modulation
                findModulationArgument = "MQAM-" + this->toString(modulationOrder);
                this->setModulation(APSKModulationBase::
                findModulation(findModulationArgument.c_str()));
        }

        /*
         * Convert and set power;
         * be sure to add required offset to ensure power is within the desired range
         */
    int power = convertStringToInt(powerString) + 4;
    this->setPower(W (1000 * pow(10, (power / 10))));

        /*
         * Convert and set bandwidth;
         * be sure to add required offset to ensure power is within the desired range
         */
    int bandwidth = this->convertStringToInt(bandwidthString) + 1;
    this->setBandwidth(Hz (bandwidth * 1000000));
}

std::string CRCENetworkRadio::toString(int numberToConvert) {
        std::stringstream numberConverter;
        numberConverter << numberToConvert;
        return numberConverter.str();
}
```

**Listing B.4. Header file for new radio type.**

```
/*
 * CRCENetworkRadio.h
 *
 * Created on: Jan 12, 2018
 * Author: Dan Hart
 */

#ifndef CRCENETWORKRADIO_H_
#define CRCENETWORKRADIO_H_

#include <math.h>
#include <sstream>
#include <string>

#include "omnetpp.h"
#include "inet/physicallayer/base/packetlevel/APSKModulationBase.h"
#include "inet/physicallayer/apskradio/packetlevel/APSKRadio.h"

#define CE_UPDATE_PARAMETERS 255

using namespace inet::physicallayer;

class CRCENetworkRadio: public APSKRadio {
        public:
                CRCENetworkRadio();
                virtual ~CRCENetworkRadio();

        int convertStringToInt(std::string binaryValue);

                std::string toString(int numberToConvert);

                void updateParameters(std::string newParameters);
};

#endif /* CRCENETWORKRADIO_H_ */
```

**Listing B.5. Auxiliary class for use with cognitive engine; stores all current population members, inserts new members in accordance with their fitness values and deletes members once the population is full.**

```
/*
 * CRCEPopulation.cpp
 *
 * Created on: Jan 11, 2018
 * Author: Dan Hart
 */

//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program.  If not, see http://www.gnu.org/licenses/.
//

#include "CRCEPopulation.h"

CRCEPopulation::CRCEPopulation(int populationSize) {
        //Set population size
        this->populationSize = populationSize;

    //Set pointers to first and last population members to null
    firstPopulationMember = NULL;
    lastPopulationMember = NULL;

}

CRCEPopulation::~CRCEPopulation() {
    // TODO Auto-generated destructor stub
}

//Method to add new member to the population
void CRCEPopulation::addMemberToPopulation(std::string newMemberEncoding,
double newMemberFitnessScore, bool initializing) {

        //Create a new population member with the encoding and fitness score provided
        CRCEPopulationMember* newMember =
        new CRCEPopulationMember(newMemberFitnessScore, newMemberEncoding);

        //int score = newMember->getFitnessScore();
        //std::string = newMember->getParameterEncoding();

        if(this->firstPopulationMember == NULL) {
                /*
                 * Add first member to the population
                 * Set both pointers to this member
                 */
                firstPopulationMember = newMember;
                lastPopulationMember = newMember;
        }
        else {
                /*
                 * Iterate through population list until the member with the next
                 * largest score as compared to the new member is found
                 * Add the member into the list, ensuring to correct pointers
```

```
 * Only update the first and last population member pointers if
 * placing new member at the front or back of the population list
 */
CRCEPopulationMember* populationListIterator =
this->firstPopulationMember;

if(populationListIterator->getNext() == NULL &&
populationListIterator->getPrevious() == NULL) {

        /*
         * Only one member in the population
         * need to explicitly check to see where new member belongs
         */
        if(populationListIterator->getFitnessScore() <
        newMember->getFitnessScore()) {

                //Add new member to the end of the list
                populationListIterator->setNext(newMember);
                newMember->setPrevious(populationListIterator);
                this->lastPopulationMember = newMember;
        }
        else {
                //Add new member to the beginning of the list
                newMember->setNext(populationListIterator);
                populationListIterator->setPrevious(newMember);
                this->firstPopulationMember = newMember;
        }
}
else {
        /*
         * If more than one entry in list, iterate until pointer
         * is pointing at member before which new member
         * should be placed
         * Second condition used to ensure we do not fall
         * off the end of the list
         */
        while((populationListIterator->getFitnessScore() >
        newMember->getFitnessScore()) &&
        !(populationListIterator->getNext() == NULL)) {

                populationListIterator =
                populationListIterator->getNext();
        }

        /*
         * Check to see if adding new member to front of list;
         * update front of list pointer if so
         */
        if(populationListIterator->getPrevious() == NULL) {
                /*
                 * Adding the new member to the front
                 * of the population list;
                 * update the pointer to the front of the list
                 */
                this->firstPopulationMember = newMember;
                populationListIterator->setPrevious(newMember);
                newMember->setNext(populationListIterator);
        }

        /*
         * While loop stopped at last entry in list
         * Must check to see if new member belongs at the end
         * of the list or as the second-to-last entry
         * by comparing fitness values
         */
        else if(populationListIterator->getNext() == NULL) {
```

```
                                        if(populationListIterator->getFitnessScore() >
                                        newMember->getFitnessScore()) {

                                                populationListIterator->setNext(newMember);
                                                newMember->setPrevious(populationListIterator);
                                                this->lastPopulationMember = newMember;
                                        }
                                        else {
                                                /*
                                                 * Add new member as
                                                 * second to last entry in the list
                                                 */
                                                newMember->
                                                setPrevious(populationListIterator->
                                                getPrevious());

                                                newMember->setNext(populationListIterator);

                                                populationListIterator->
                                                getPrevious()->setNext(newMember);

                                                populationListIterator->
                                                setPrevious(newMember);
                                        }
                                }

                                else {
                                        /*
                                         * Adding new member to middle of the list;
                                         * do not need to update pointers to front
                                         * or back of the list
                                         */
                                        newMember->
                                        setPrevious(populationListIterator->getPrevious());

                                        newMember->setNext(populationListIterator);

                                        populationListIterator->
                                        getPrevious()->setNext(newMember);

                                        populationListIterator->setPrevious(newMember);
                                }
                        }
                }

                //If not initializing, keep population constrained by eliminating last member
                if(!initializing) {
                        this->lastPopulationMember =
                        this->lastPopulationMember->getPrevious();

                        this->lastPopulationMember->setNext(NULL);
                }
        }

        //Method to return chromosome from specified population member
        std::string CRCEPopulation::getPopulationMember(int populationMember) {
                //Get pointer to first member in the population
                CRCEPopulationMember* nextPopulationMember = this->firstPopulationMember;

                /*
                 * Iterate through population members
                 * returning the chromosome for the specified member
                 */
                for(int index = 1; index < populationMember; index++) {
                        nextPopulationMember = nextPopulationMember->getNext();
                }
```

```
        return nextPopulationMember->getParameterEncoding();
}
```

**Listing B.6. Header file for population class.**

```
/*
 * CRCEPopulation.h
 *
 * Created on: Jan 11, 2018
 * Author: Dan Hart
 */

//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program.  If not, see http://www.gnu.org/licenses/.
//

#ifndef CRCEPOPULATION_H_
#define CRCEPOPULATION_H_

#include "CRCEPopulationMember.h"
#include <iostream>
#include <string>

class CRCEPopulation {
    public:
        CRCEPopulation(int populationSize);
        virtual ~CRCEPopulation();

        //VARIABLES

        //Reference to first member in population
        CRCEPopulationMember* firstPopulationMember;

        //Tracks last member in the population
        CRCEPopulationMember* lastPopulationMember;

        //Dictates population size
        int populationSize;



        //METHODS

        //Used to add a member to the population
        void addMemberToPopulation(std::string newMemberEncoding,
        double newMemberFitnessScore, bool initializing);

        //Returns reference to specified population member
        std::string getPopulationMember(int populationMember);
};

#endif /* CRCEPOPULATION_H_ */
```

**Listing B.7. Auxiliary class for use with Cognitive Engine and population; represents a single population member and stores the member's fitness value and parameter encoding.**

```
/*
 * CRCEPopulationMember.cpp
 *
 * Created on: Jan 11, 2018
 * Author: Dan Hart
 */

//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program.  If not, see http://www.gnu.org/licenses/.
//

#include "CRCEPopulationMember.h"

CRCEPopulationMember::CRCEPopulationMember(double newFitnessScore,
std::string newParameterEncoding) {

        //Don't yet have previous/next population members, make pointers null
        this->nextPopulationMember = NULL;
        this->previousPopulationMember = NULL;

        //Set population member's encoding/fitness score to values provided
        this->fitnessScore = newFitnessScore;
        this->parameterEncoding = newParameterEncoding;
}

CRCEPopulationMember::~CRCEPopulationMember() {
    // TODO Auto-generated destructor stub
}

//Method to compare two population members
bool CRCEPopulationMember::
compareConfigurations(CRCEPopulationMember candidateConfiguration) {

    if(this->fitnessScore < candidateConfiguration.getFitnessScore()) {
        return true;
    }
    return false;
}

double CRCEPopulationMember::getFitnessScore() {
    return this->fitnessScore;
}

//Getter method to return next population member
CRCEPopulationMember* CRCEPopulationMember::getNext() {
    return this->nextPopulationMember;
}

//Getter method to return the member's parameter encoding
std::string CRCEPopulationMember::getParameterEncoding() {
        return this->parameterEncoding;
```

```
}

//Getter method to return previous population member
CRCEPopulationMember* CRCEPopulationMember::getPrevious() {
    return this->previousPopulationMember;
}

//Setter method to set next population member
void CRCEPopulationMember::setNext(CRCEPopulationMember* nextMember) {
    this->nextPopulationMember = nextMember;
}

//Setter method to set previous population member
void CRCEPopulationMember::setPrevious(CRCEPopulationMember* previousMember) {
    this->previousPopulationMember = previousMember;
}
```

**Listing B.8. Header file for population member class.**

```
/*
 * CRCEPopulationMember.h
 *
 * Created on: Jan 11, 2018
 * Author: Dan Hart
 */

//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program.  If not, see http://www.gnu.org/licenses/.
//

#ifndef CRCEPOPULATIONMEMBER_H_
#define CRCEPOPULATIONMEMBER_H_

#include <string>

class CRCEPopulationMember {
    public:
        //Class constructors and destructor
        CRCEPopulationMember(double fitnessScore, std::string newParameterEncoding);
        virtual ~CRCEPopulationMember();

        //VARIABLES

        //Variable to hold population member's fitness score
        double fitnessScore;

        //Pointer referencing the next member in the population
        CRCEPopulationMember* nextPopulationMember;

        //String to hold the population member's parameter encoding
        std::string parameterEncoding;

        //Pointer referencing the previous member in the population
        CRCEPopulationMember* previousPopulationMember;



        //METHODS

        //Method to compare two population members
        bool compareConfigurations(CRCEPopulationMember candidateConfiguration);

        //Getter method to return a given population member's fitness score
        double getFitnessScore();

        //Getter method to return next population member
        CRCEPopulationMember* getNext();

        //Getter method to return the member's parameter encoding
        std::string getParameterEncoding();

        //Getter method to return previous population member
```

```
        CRCEPopulationMember* getPrevious();

        //Setter method to set next population member
        void setNext(CRCEPopulationMember* nextMember);

        //Setter method to set previous population member
        void setPrevious(CRCEPopulationMember* previousMember);
};

#endif /* CRCEPOPULATIONMEMBER_H_ */
```

# Appendix C. Code Listings for MATLAB Engine Implementation

**Listing C.1. MATLAB code listing for main program run to evaluate fitness function; evaluation of the 30 runs for a given test case occur in parallel.**

```
%%Clean up
clear all;
close all;
clc;

%% Engine settings

alpha = .33;
beta = .33;
bitrate = 500000;           %bps
chromosomeLength = 15;      %bits
codeNumber = 0;
codingRate = 1;
datagramHeaderSize = 34;    %bytes
frameHeaderSize = 30;       %bytes
frameSize = 4905;           %bytes
lambda = .33;
latencyWeight = .8;
maxBandwidth = 32;          %MHz
maxOrder = 8;
maxPower = 35;              %dBm
minBandwidth = 1;           %MHz
minOrder = 1;
minPower = 4;               %dBm
packetHeaderSize = 20;
packetSizeBytes = 1050000;
populationSize = 50;
powerWeight = (1 - latencyWeight) / 2;
TDD = 1;
throughputWeight = (1 - latencyWeight) / 2;

%% Run genetic algorithm
%Run algorithm 30 times at each setting for the noise floor and number of
%generations
%Save the returned fitness member, their fitness value and the runtime

encodingSchemeSize = 270;
noiseFloorSize = 90;
numberGenerationsSize = 30;

%Create arrays in which to store results
numberOfGenerationsResults = zeros(270, 6);

%Create for loop for -50 dBm noise floor and number of generations of 200
noiseFloor = -50;
numberOfGenerations = 200;
offset = 0;
for index=1:30
    resultsIterator(index) = parfeval(@cognitiveEngine, 6, alpha, beta, bitrate,...
        chromosomeLength, codeNumber, codingRate, datagramHeaderSize, frameHeaderSize,...
        frameSize, lambda, latencyWeight, maxBandwidth, maxOrder, maxPower, minBandwidth,...
        minOrder, minPower, noiseFloor, numberOfGenerations, packetHeaderSize,...
        packetSizeBytes, populationSize, powerWeight, TDD, throughputWeight);
end

for index2 = 1:30
    [runNum, resultFitness, resultModScheme, resultModOrder, resultPower,...
        resultBandwidth, runtime] = fetchNext(resultsIterator);
```

```
    numberOfGenerationsResults(runNum + offset, 1) = resultFitness;
    numberOfGenerationsResults(runNum + offset, 2) = resultModScheme;
    numberOfGenerationsResults(runNum + offset, 3) = resultModOrder;
    numberOfGenerationsResults(runNum + offset, 4) = resultPower;
    numberOfGenerationsResults(runNum + offset, 5) = resultBandwidth;
    numberOfGenerationsResults(runNum + offset, 6) = runtime;
end



%Create for loop for -50 dBm noise floor and number of generations of 600
noiseFloor = -50;
numberOfGenerations = 600;
offset = 30;
for index=1:30
    resultsIterator(index) = parfeval(@cognitiveEngine, 6, alpha, beta, bitrate,...
        chromosomeLength, codeNumber, codingRate, datagramHeaderSize, frameHeaderSize,...
        frameSize, lambda, latencyWeight, maxBandwidth, maxOrder, maxPower, minBandwidth,...
        minOrder, minPower, noiseFloor, numberOfGenerations, packetHeaderSize,...
        packetSizeBytes, populationSize, powerWeight, TDD, throughputWeight);
end

for index2 = 1:30
    [runNum, resultFitness, resultModScheme, resultModOrder, resultPower,...
        resultBandwidth, runtime] = fetchNext(resultsIterator);
    numberOfGenerationsResults(runNum + offset, 1) = resultFitness;
    numberOfGenerationsResults(runNum + offset, 2) = resultModScheme;
    numberOfGenerationsResults(runNum + offset, 3) = resultModOrder;
    numberOfGenerationsResults(runNum + offset, 4) = resultPower;
    numberOfGenerationsResults(runNum + offset, 5) = resultBandwidth;
    numberOfGenerationsResults(runNum + offset, 6) = runtime;
end



%Create for loop for -50 dBm noise floor and number of generations of 1000
noiseFloor = -50;
numberOfGenerations = 1000;
offset = 60;
for index=1:30
resultsIterator(index) = parfeval(@cognitiveEngine, 6, alpha, beta, bitrate,...
        chromosomeLength, codeNumber, codingRate, datagramHeaderSize, frameHeaderSize,...
        frameSize, lambda, latencyWeight, maxBandwidth, maxOrder, maxPower, minBandwidth,...
        minOrder, minPower, noiseFloor, numberOfGenerations, packetHeaderSize,...
        packetSizeBytes, populationSize, powerWeight, TDD, throughputWeight);
end

for index2 = 1:30
    [runNum, resultFitness, resultModScheme, resultModOrder, resultPower,...
        resultBandwidth, runtime] = fetchNext(resultsIterator);
    numberOfGenerationsResults(runNum + offset, 1) = resultFitness;
    numberOfGenerationsResults(runNum + offset, 2) = resultModScheme;
    numberOfGenerationsResults(runNum + offset, 3) = resultModOrder;
    numberOfGenerationsResults(runNum + offset, 4) = resultPower;
    numberOfGenerationsResults(runNum + offset, 5) = resultBandwidth;
    numberOfGenerationsResults(runNum + offset, 6) = runtime;
end



%Create for loop for -30 dBm noise floor and number of generations of 200
noiseFloor = -30;
numberOfGenerations = 200;
offset = 90;
for index=1:30
resultsIterator(index) = parfeval(@cognitiveEngine, 6, alpha, beta, bitrate,...
```

```
        chromosomeLength, codeNumber, codingRate, datagramHeaderSize, frameHeaderSize,...
        frameSize, lambda, latencyWeight, maxBandwidth, maxOrder, maxPower, minBandwidth,...
        minOrder, minPower, noiseFloor, numberOfGenerations, packetHeaderSize,...
        packetSizeBytes, populationSize, powerWeight, TDD, throughputWeight);
end

for index2 = 1:30
    [runNum, resultFitness, resultModScheme, resultModOrder, resultPower,...
        resultBandwidth, runtime] = fetchNext(resultsIterator);
    numberOfGenerationsResults(runNum + offset, 1) = resultFitness;
    numberOfGenerationsResults(runNum + offset, 2) = resultModScheme;
    numberOfGenerationsResults(runNum + offset, 3) = resultModOrder;
    numberOfGenerationsResults(runNum + offset, 4) = resultPower;
    numberOfGenerationsResults(runNum + offset, 5) = resultBandwidth;
    numberOfGenerationsResults(runNum + offset, 6) = runtime;
end



%Create for loop for -30 dBm noise floor and number of generations of 600
noiseFloor = -30;
numberOfGenerations = 600;
offset = 120;
for index=1:30
resultsIterator(index) = parfeval(@cognitiveEngine, 6, alpha, beta, bitrate,...
        chromosomeLength, codeNumber, codingRate, datagramHeaderSize, frameHeaderSize,...
        frameSize, lambda, latencyWeight, maxBandwidth, maxOrder, maxPower, minBandwidth,...
        minOrder, minPower, noiseFloor, numberOfGenerations, packetHeaderSize,...
        packetSizeBytes, populationSize, powerWeight, TDD, throughputWeight);
end

for index2 = 1:30
    [runNum, resultFitness, resultModScheme, resultModOrder, resultPower,...
        resultBandwidth, runtime] = fetchNext(resultsIterator);
    numberOfGenerationsResults(runNum + offset, 1) = resultFitness;
    numberOfGenerationsResults(runNum + offset, 2) = resultModScheme;
    numberOfGenerationsResults(runNum + offset, 3) = resultModOrder;
    numberOfGenerationsResults(runNum + offset, 4) = resultPower;
    numberOfGenerationsResults(runNum + offset, 5) = resultBandwidth;
    numberOfGenerationsResults(runNum + offset, 6) = runtime;
end



%Create for loop for -30 dBm noise floor and number of generations of 1000
noiseFloor = -30;
numberOfGenerations = 1000;
offset = 150;
for index=1:30
resultsIterator(index) = parfeval(@cognitiveEngine, 6, alpha, beta, bitrate,...
        chromosomeLength, codeNumber, codingRate, datagramHeaderSize, frameHeaderSize,...
        frameSize, lambda, latencyWeight, maxBandwidth, maxOrder, maxPower, minBandwidth,...
        minOrder, minPower, noiseFloor, numberOfGenerations, packetHeaderSize,...
        packetSizeBytes, populationSize, powerWeight, TDD, throughputWeight);
end

for index2 = 1:30
    [runNum, resultFitness, resultModScheme, resultModOrder, resultPower,...
        resultBandwidth, runtime] = fetchNext(resultsIterator);
    numberOfGenerationsResults(runNum + offset, 1) = resultFitness;
    numberOfGenerationsResults(runNum + offset, 2) = resultModScheme;
    numberOfGenerationsResults(runNum + offset, 3) = resultModOrder;
    numberOfGenerationsResults(runNum + offset, 4) = resultPower;
    numberOfGenerationsResults(runNum + offset, 5) = resultBandwidth;
    numberOfGenerationsResults(runNum + offset, 6) = runtime;
end
```

```
%Create for loop for 10 dBm noise floor and number of generations of 200
noiseFloor = 10;
numberOfGenerations = 200;
offset = 180;
for index=1:30
resultsIterator(index) = parfeval(@cognitiveEngine, 6, alpha, beta, bitrate,...
        chromosomeLength, codeNumber, codingRate, datagramHeaderSize, frameHeaderSize,...
        frameSize, lambda, latencyWeight, maxBandwidth, maxOrder, maxPower, minBandwidth,...
        minOrder, minPower, noiseFloor, numberOfGenerations, packetHeaderSize,...
        packetSizeBytes, populationSize, powerWeight, TDD, throughputWeight);end

for index2 = 1:30
    [runNum, resultFitness, resultModScheme, resultModOrder, resultPower,...
        resultBandwidth, runtime] = fetchNext(resultsIterator);
    numberOfGenerationsResults(runNum + offset, 1) = resultFitness;
    numberOfGenerationsResults(runNum + offset, 2) = resultModScheme;
    numberOfGenerationsResults(runNum + offset, 3) = resultModOrder;
    numberOfGenerationsResults(runNum + offset, 4) = resultPower;
    numberOfGenerationsResults(runNum + offset, 5) = resultBandwidth;
    numberOfGenerationsResults(runNum + offset, 6) = runtime;
end



%Create for loop for 10 dBm noise floor and number of generations of 600
noiseFloor = 10;
numberOfGenerations = 600;
offset = 210;
for index=1:30
resultsIterator(index) = parfeval(@cognitiveEngine, 6, alpha, beta, bitrate,...
        chromosomeLength, codeNumber, codingRate, datagramHeaderSize, frameHeaderSize,...
        frameSize, lambda, latencyWeight, maxBandwidth, maxOrder, maxPower, minBandwidth,...
        minOrder, minPower, noiseFloor, numberOfGenerations, packetHeaderSize,...
        packetSizeBytes, populationSize, powerWeight, TDD, throughputWeight);
end

for index2 = 1:30
    [runNum, resultFitness, resultModScheme, resultModOrder, resultPower,...
        resultBandwidth, runtime] = fetchNext(resultsIterator);
    numberOfGenerationsResults(runNum + offset, 1) = resultFitness;
    numberOfGenerationsResults(runNum + offset, 2) = resultModScheme;
    numberOfGenerationsResults(runNum + offset, 3) = resultModOrder;
    numberOfGenerationsResults(runNum + offset, 4) = resultPower;
    numberOfGenerationsResults(runNum + offset, 5) = resultBandwidth;
    numberOfGenerationsResults(runNum + offset, 6) = runtime;
end



%Create for loop for 10 dBm noise floor and number of generations of 1000
noiseFloor = 10;
numberOfGenerations = 1000;
offset = 240;
for index=1:30
resultsIterator(index) = parfeval(@cognitiveEngine, 6, alpha, beta, bitrate,...
        chromosomeLength, codeNumber, codingRate, datagramHeaderSize, frameHeaderSize,...
        frameSize, lambda, latencyWeight, maxBandwidth, maxOrder, maxPower, minBandwidth,...
        minOrder, minPower, noiseFloor, numberOfGenerations, packetHeaderSize,...
        packetSizeBytes, populationSize, powerWeight, TDD, throughputWeight);
end

for index2 = 1:30
    [runNum, resultFitness, resultModScheme, resultModOrder, resultPower,...
```

```
        resultBandwidth, runtime] = fetchNext(resultsIterator);
    numberOfGenerationsResults(runNum + offset, 1) = resultFitness;
    numberOfGenerationsResults(runNum + offset, 2) = resultModScheme;
    numberOfGenerationsResults(runNum + offset, 3) = resultModOrder;
    numberOfGenerationsResults(runNum + offset, 4) = resultPower;
    numberOfGenerationsResults(runNum + offset, 5) = resultBandwidth;
    numberOfGenerationsResults(runNum + offset, 6) = runtime;
end


%Update .mat file name to match cases being run (e.g., FEC encoded or not)
save EngineResultsUpdatedSNRNoFECEncodingParallelRun numberOfGenerationsResults...
    encodingSchemeSize noiseFloorSize generationSize;
```

**Listing C.2. MATLAB code listing for CE function written in order to parallelize execution; main file calls this function in parallel and saves results.**

```
function [resultFitness, resultModScheme, resultModOrder, resultPower,...
    resultBandwidth, runtime] =...
    cognitiveEngine(alpha, beta, bitrate, chromosomeLength, codeNumber,...
    codingRate, datagramHeaderSize, frameHeaderSize, frameSize, lambda,...
    latencyWeight, maxBandwidth, maxOrder, maxPower, minBandwidth, minOrder,...
    minPower, noiseFloor, numberOfGenerations, packetHeaderSize, packetSizeBytes,...
    populationSize, powerWeight, TDD, throughputWeight);

%Create and initialize population
population = zeros(populationSize, 5);
populationArraySize = size(population);
population(2:populationArraySize(1), 1:1) = -1;

%Due to the way in which the insert into function works, must keep the
%first row as a placeholder
population(1,1) = 1;

for index = 1:populationSize
    [fitnessScore, modulationScheme, modulationOrder, power, bandwidth] =...
        generateRandomPopulationMember(alpha, beta, bitrate, codingRate,...
        datagramHeaderSize, frameHeaderSize, frameSize, lambda, latencyWeight,...
        maxBandwidth, maxOrder, maxPower, minBandwidth, minOrder, minPower,...
        noiseFloor, packetHeaderSize, packetSizeBytes, powerWeight, TDD,...
        throughputWeight);
    newMember = [fitnessScore, modulationScheme, modulationOrder, power, bandwidth];
    population = insertMemberIntoPopulation(newMember, population, populationSize);
end

tic
[resultFitness, resultModScheme, resultModOrder, resultPower, resultBandwidth] =...
    runGeneticAlgorithm(alpha, beta, bitrate, chromosomeLength, codeNumber,...
    codingRate, datagramHeaderSize, frameHeaderSize, frameSize, lambda,...
    latencyWeight, maxBandwidth, maxOrder, maxPower, minOrder, noiseFloor,...
    numberOfGenerations, packetHeaderSize, packetSizeBytes, population,...
    populationSize, powerWeight, TDD, throughputWeight);
runtime = toc;
```

**Listing C.3. MATLAB code listing for generating a random population member.**

```
%Function to generate a random population member
%Will be called by population initialization function
function [fitnessScore, modulationScheme, modulationOrder, power, bandwidth] =...
    generateRandomPopulationMember(alpha, beta, bitrate, codingRate,...
    datagramHeaderSize, frameHeaderSize, frameSize, lambda, latencyWeight,...
    maxBandwidth, maxOrder, maxPower, minBandwidth, minOrder, minPower,...
    noiseFloor, packetHeaderSize, packetSizeBytes, powerWeight, TDD, throughputWeight)

    modulationScheme = generateModulationScheme();

    modulationOrder = 2 ^ generateModulationOrder(minOrder, maxOrder, modulationScheme == 1);

    power = generatePower(minPower, maxPower);

    bandwidth = generateBandwidth(minBandwidth, maxBandwidth);

    fitnessScore =...
        computeFitnessScore(alpha, bandwidth, beta, bitrate, 0, codingRate,...
        datagramHeaderSize, frameHeaderSize, frameSize, lambda, latencyWeight,...
        maxBandwidth, maxOrder, maxPower, minOrder, modulationOrder, noiseFloor,...
        packetHeaderSize, packetSizeBytes, power, powerWeight, TDD,...
        throughputWeight, modulationScheme == 1);
end

%Method to generate a random bandwidth; used in initializing the population
function bandwidth = generateBandwidth(minBandwidth, maxBandwidth)
        possibleBandwidths = minBandwidth:1:maxBandwidth;
    possibleBandwidthsSize = size(possibleBandwidths);
    chosenBandwidth = randi([1 possibleBandwidthsSize(2)], 1, 1);
    bandwidth = possibleBandwidths(chosenBandwidth(1));
end

%Method to generate a random modulation scheme encoding;
%used in initializing the population
function modulationScheme = generateModulationScheme()
    chosenScheme = randi([0 1], 1, 1);
    modulationScheme = chosenScheme(1);
end

%Method to generate a random modulation order encoding;
%will be used in initializing the population
function modulationOrder = generateModulationOrder(minOrder, maxOrder, usingQAM)
        %If using QAM, return random, even modulation order
    if usingQAM
        evenModulationOrders = 2:2:maxOrder;
        evenModulationOrdersSize = size(evenModulationOrders);
        chosenOrder = randi([1 evenModulationOrdersSize(2)], 1, 1);
        modulationOrder = evenModulationOrders(chosenOrder(1));
    else
        modulationOrders = minOrder:1:maxOrder;
        modulationOrdersSize = size(modulationOrders);
        chosenOrder = randi([1 modulationOrdersSize(2)], 1, 1);
        modulationOrder = modulationOrders(chosenOrder(1));
    end
end

%Method to generate a random modulation power encoding;
%used in initializing the population
function power = generatePower(minPower, maxPower)
        possiblePowers = minPower:1:maxPower;
    possiblePowersSize = size(possiblePowers);
    chosenPower = randi([1 possiblePowersSize(2)], 1, 1);
    power = possiblePowers(chosenPower(1));
end
```

**Listing C.4. MATLAB code listing for evaluating fitness function.**

```
%%Function to compute fitness score; uses other helper methods
function fitnessScore =...
    computeFitnessScore(alpha, bandwidth, beta, bitrate, codeNumber, codingRate,...
    datagramHeaderSize, frameHeaderSize, frameSize, lambda, latencyWeight,...
    maxBandwidth, maxOrder, maxPower, minOrder, modulationOrder, noiseFloor,...
    packetHeaderSize, packetSizeBytes, power, powerWeight, TDD,...
    throughputWeight, usingQAM)

    %Parameter settings in OMNet program subsumed in BER
    %function
    SNR = power - noiseFloor;
    bitErrorRate = computeBER(packetSizeBytes, modulationOrder, SNR, usingQAM,...
        codeNumber);

    %Calculate the individual fitness score components
    latencyFitness = computeLatencyFitness(bitErrorRate, frameHeaderSize,...
        frameSize);
    powerFitness =...
        computePowerFitness(alpha, bandwidth, beta, bitrate, lambda, maxBandwidth,...
        maxOrder, maxPower, minOrder, modulationOrder, power);
    throughputFitness =...
        computeThroughputFitness(bitErrorRate, codingRate, datagramHeaderSize,...
        frameHeaderSize, frameSize, packetHeaderSize, TDD);

    %Multiply each component by its associated weight and set result
    fitnessScore = latencyWeight * latencyFitness + powerWeight * powerFitness +...
        throughputWeight * throughputFitness;
end

%%Function to compute the latency component of the fitness score
function latencyFitness = computeLatencyFitness(bitErrorRate, frameHeaderSize, frameSize)
        latencyFitness = 1 - (1 - ((1 - bitErrorRate) ^ ((frameSize + frameHeaderSize) * 8)));
end

%%Function to compute
function powerFitness =...
    computePowerFitness(alpha, bandwidth, beta, bitrate, lambda, maxBandwidth,...
    maxOrder, maxPower, minOrder, modulationOrder, power)
        %Calculate the three components individually to try and
    %keep things a little more readable/manageable
        componentOne = alpha * (((maxPower + maxBandwidth) - (power + bandwidth)) /...
        (maxPower + maxBandwidth));
        componentTwo = beta * ((maxOrder - log2(modulationOrder)) / maxOrder);
        componentThree = lambda * (((bitrate / minOrder) - (bitrate /...
        log2(modulationOrder))) / (bitrate / minOrder));

        %Return the sum of the three components
        powerFitness = componentOne + componentTwo + componentThree;
end

%Function to compute the throughput component of the fitness score
function throughputFitness =...
    computeThroughputFitness(bitErrorRate, codingRate, datagramHeaderSize,...
    frameHeaderSize, frameSize, packetHeaderSize, TDD)

        throughputFitness =...
        (frameSize / (frameSize + frameHeaderSize + packetHeaderSize +...
        datagramHeaderSize)) * (1 - bitErrorRate) ^ (frameSize + frameHeaderSize)...
        * codingRate * TDD;
end
```

**Listing C.5. MATLAB code listing for computing bit error rate.**

```
%Function to generate and then FEC encode, modulate, add noise to,
%transmit, decode and demodulate the signal and return error information
%about the received signal
function [BER, numberErroredBits, totalBits] =...
    computeBER(packetSizeBytes, modulationOrder, SNR, usingQAM, codeNumber)
    %Use persistent txBits variable (will "transmit" the same bits each time
    persistent txBits;

    %Create bits to be transmitted
    txBits = randi([0 1], packetSizeBytes * 8, 1);

    %Selectively FEC Encode and then modulate signal
    if(codeNumber > 0)
        codedBits = fecEncode(txBits, modulationOrder, codeNumber);
        txSignal = generateTXSignal(codedBits, modulationOrder, usingQAM);
    else
        txSignal = generateTXSignal(txBits, modulationOrder, usingQAM);
    end

    %Add noise to the signal ("transmit" signal)
    noisySignal = simulateChannel(SNR, txSignal);

    %Demodulate "received" signal
    rxBits = demodulateSignal(noisySignal, modulationOrder, usingQAM);

    %FEC Decode signal
    if(codeNumber > 0)
        rxBits = fecDecode(rxBits, modulationOrder, codeNumber);
    end

    %Collect error statistics on received signal
    if(codeNumber > 0)
        hErrorRate = comm.ErrorRate('ComputationDelay', 3);
    else
        hErrorRate = comm.ErrorRate;
    end

    errorStats = hErrorRate(txBits, rxBits);
    BER = errorStats(1);
    numberErroredBits = errorStats(2);
    totalBits = errorStats(3);
end

%Function to encode bits according to selected encoder
function txBitsEncoded = fecEncode(txBits, modulationOrder, codeSelector)
    switch codeSelector
        case 1
            encoder = comm.BCHEncoder;
            txBitsEncoded = encoder(txBits);
%         case 2
%             %Cyclic block encoding
%             txBitsEncoded = encode(txBits, n, k, 'cyclic/binary');
%         case 3
%             %Linear block encoding
%             txBitsEncoded = encode(txBits, n, k, 'linear/binary');
%         case 4
%             %Convolutional coding
%             trellis = struct('numInputSymbols', modulationOrder, 'numOutputSymbols',...
%                 2 * modulationOrder, 'nextStates', [0 2 ; 0 2 ; 1 3 ; 1 3],...
%                 'outputs', [0 3 ; 1 2 ; 3 0 ; 2 1]);
%             txBitsEncoded = convenc(txBits, trellis);
    end
end
```

```
%Function to modulate bits to be transmitted according to modulator type
%and order
function txSignal = generateTXSignal(txBits, modulationOrder, usingQAM)
    %Check to ensure the appropriate modulator is used
    if usingQAM
        hQAMModulator = comm.RectangularQAMModulator('NormalizationMethod',...
            'Average power', 'AveragePower', 1, 'ModulationOrder',...
            modulationOrder, 'BitInput', true);
        txSignal = (hQAMModulator(txBits));
    else
        hPSKModulator = comm.PSKModulator('ModulationOrder', modulationOrder,...
            'BitInput', true);
        txSignal = (hPSKModulator(txBits));
    end
end

%Function to add noise to the signal, simulating signal transmission
function [noisySignal] = simulateChannel(SNR, txSignal)
    %Create channel
    hAWGN = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
        'SNR', SNR);
    %Add noise to signal ("transmit" signal)
    noisySignal = hAWGN(txSignal);
end

%Function to demodulate received signal
function [rxBits] = demodulateSignal(noisySignal, modulationOrder, usingQAM)
    %Use global PSK and QAM demodulator objects
    %Check to ensure appropriate demodulator is used
    if usingQAM
        hQAMDemodulator = comm.RectangularQAMDemodulator('NormalizationMethod',...
            'Average power', 'AveragePower', 1, 'ModulationOrder', modulationOrder,...
            'BitOutput', true);
        rxBits = hQAMDemodulator(noisySignal);
    else
        hPSKDemodulator = comm.PSKDemodulator('ModulationOrder', modulationOrder,...
            'BitOutput', true);
        rxBits = hPSKDemodulator(noisySignal);
    end
end

%Function to decode bits according to selected encoder
function rxBits = fecDecode(rxBitsEncoded, modulationOrder, codeNumber)
    switch codeNumber
        case 1
            decoder = comm.BCHDecoder;
            rxBits = decoder(rxBitsEncoded);
%         case 2
%             %Cyclic block encoding
%             rxBits = decode(rxBitsEncoded, n, k, 'cyclic/binary');
%         case 3
%             %Linear block encoding
%             rxBits = decode(rxBitsEncoded, n, k, 'linear/binary');
%         case 4
%             %Convolutional coding
%             trellis = struct('numInputSymbols', modulationOrder, 'numOutputSymbols',...
%                 2 * modulationOrder, 'nextStates', [0 2 ; 0 2 ; 1 3 ; 1 3],...
%                 'outputs', [0 3 ; 1 2 ; 3 0 ; 2 1]);
%             rxBits = vitdec(rxBitsEncoded, trellis, 64, 'trunc', 'hard');
    end
end
```

**Listing C.6. MATLAB code listing for inserting new member into the population.**

```
function population =...
    insertMemberIntoPopulation(newMember, population, populationSize)

    %Add new member to the population

    index = 1;
    %Find the appropriate place for the new member to be inserted
    while (population(index, 1) > newMember(1) && index <= populationSize)
        index = index + 1;
    end

    %If not at the end of the population, insert new member at index2
    %If at end of population, check to see if new member should be
    %added as last population member, otherwise don't add (not fit
    %enough)
    if ((index < populationSize) || (index == populationSize && newMember(1)...
            >= population(index, 1)))
        %insertrows function came from MATLAB community forum; website:

        %"https://www.mathworks.com/matlabcentral/fileexchange/9984-
        %insertrows-a-b-ind-"

        population = insertrows(population, newMember, index - 1);
    end
end
```

**Listing C.7. MATLAB code listing for genetic algorithm.**

```
%Runs the genetic algorithm
function [newFitnessValue, modulationScheme, modulationOrder, power, bandwidth] =...
    runGeneticAlgorithm(alpha, beta, bitrate, chromosomeLength, codeNumber,...
    codingRate, datagramHeaderSize, frameHeaderSize, frameSize, lambda, latencyWeight,...
    maxBandwidth, maxOrder, maxPower, minOrder, noiseFloor, numberOfGenerations,...
    packetHeaderSize, packetSizeBytes, population, populationSize, powerWeight,...
    TDD, throughputWeight)

    generations = 1:1:numberOfGenerations;

    for index = generations
            %Randomly choose two chromosomes from population for recombination
        randomNumbers = randi([1 populationSize], 1, 2);
        firstChromosome = randomNumbers(1);
        secondChromosome = randomNumbers(2);

        %Recombine and mutate population members
        %Extract population members from population
        firstModScheme = population(firstChromosome, 2);
        firstModOrder = population(firstChromosome, 3);
        firstPower = population(firstChromosome, 4);
        firstBandwidth = population(firstChromosome, 5);

        secondModScheme = population(secondChromosome, 2);
        secondModOrder = population(secondChromosome, 3);
        secondPower = population(secondChromosome, 4);
        secondBandwidth = population(secondChromosome, 5);

        firstMember = [firstModScheme firstModOrder firstPower firstBandwidth];
        secondMember = [secondModScheme secondModOrder secondPower secondBandwidth];

        [recombinedModScheme, recombinedModOrder, recombinedPower, recombinedBandwidth] =...
            recombineChromosomes(firstMember, secondMember);

        recombinedChromosome =...
            [recombinedModScheme recombinedModOrder recombinedPower recombinedBandwidth];

                [mutatedModScheme, mutatedModOrder, mutatedPower, mutatedBandwidth] =...
            mutateChromosome(recombinedChromosome, chromosomeLength);

        mutatedChromosome = [mutatedModScheme mutatedModOrder mutatedPower mutatedBandwidth];

        modulationScheme = mutatedChromosome(1);
        modulationOrder = 2 ^ mutatedChromosome(2);
        power = mutatedChromosome(3) + 4;
        bandwidth = mutatedChromosome(4) + 1;

        %Compute the new member's fitness value
                newFitnessValue = ...
            computeFitnessScore(alpha, bandwidth, beta, bitrate, codeNumber,...
            codingRate, datagramHeaderSize, frameHeaderSize, frameSize, lambda,...
            latencyWeight, maxBandwidth, maxOrder, maxPower, minOrder, modulationOrder,...
            noiseFloor, packetHeaderSize, packetSizeBytes, power, powerWeight, TDD,...
            throughputWeight, modulationScheme == 1);

        %Create a new member and add to the population
        newMember = [newFitnessValue modulationScheme modulationOrder power bandwidth];
        insertMemberIntoPopulation(newMember, population, populationSize);
    end

    %Return the new, most-fit population member
    newFitnessValue = population(2, 1);
    modulationScheme = population(2, 2);
    modulationOrder = population(2, 3);
```

```
    power = population(2, 4);
    bandwidth = population(2, 5);
end

function [mutatedModScheme, mutatedModOrder, mutatedPower, mutatedBandwidth] =...
    mutateChromosome(recombinedChromosome, chromosomeLength)
    %Generate random number to determine if each bit should be flipped
    randomNumbers = randi([0 chromosomeLength], 1, chromosomeLength);

    %Mutate modulation scheme
    if(recombinedChromosome(1) == 0)
        %Used PSK
        if(randomNumbers(1) == chromosomeLength)
            %Flip bit
            mutatedModScheme = 1;
        else
            %Don't flip bit
            mutatedModScheme = 0;
        end
    else
        %Used QAM
        if(randomNumbers(1) == chromosomeLength)
            %Flip bit
            mutatedModScheme = 0;
        else
            %Don't flip bit
            mutatedModScheme = 1;
        end
    end

        %Mutate modulation order
    modSchemeOffset = 1;
    bits = 1:1:4;
    mutatedModOrder = 0;
    for index = bits
        %Determine if bit is a one or zero
        maxPowerOfTwo = 3;
        index2 = 1;
        result = recombinedChromosome(2);
        while index2 < index
            if result >= 2 ^ (maxPowerOfTwo - (index2 - 1))
                result = result - 2 ^ (maxPowerOfTwo - (index2 - 1));
            end
            index2 = index2 + 1;
        end

        %If result is greater than or equal to 2 raised to index power, had
        %a 1
        if result >= 2 ^ (maxPowerOfTwo - (index - 1))
            if randomNumbers(index + modSchemeOffset) == chromosomeLength
            else
                %Add 2 to index power to modulation order
                mutatedModOrder = mutatedModOrder + 2 ^ (maxPowerOfTwo - (index - 1));
            end
        else
            %Had a zero
            if randomNumbers(index + modSchemeOffset) == chromosomeLength
                mutatedModOrder = mutatedModOrder + 2 ^ (maxPowerOfTwo - (index - 1));
            end
        end
    end

    %Verify modulation order is acceptable
    %If it is greater than 8, reset to 8
    %For QAM modulation, if order is odd, replace with randomly chosen even
    %order
```

```
if mutatedModOrder > 8
    mutatedModOrder = 8;
elseif mutatedModOrder == 0
    modulationOrders = 1:1:8;
    newOrder = randi([1 8], 1, 1);
    mutatedModOrder = modulationOrders(newOrder(1));
elseif (mutatedModScheme == 1) && (mod(mutatedModOrder, 2) ~= 0)
    evenModulationOrders = [2, 4, 6, 8];
    newOrder = randi([1 4], 1, 1);
    mutatedModOrder = evenModulationOrders(newOrder(1));
end

%Mutate power and bandwidth

%Mutate power
mutatedPower = 4;
modSchemeOrderOffset = 5;
powerBits = 1:1:5;
for index = powerBits
    %Determine if bit is a one or zero
    maxPowerOfTwo = 4;
    index2 = 1;
    result = recombinedChromosome(3) - 4;
    while index2 < index
        if result >= 2 ^ (maxPowerOfTwo - (index2 - 1))
            result = result - 2 ^ (maxPowerOfTwo - (index2 - 1));
        end
        index2 = index2 + 1;
    end

    %If result is greater than or equal to 2 raised to index power, had
    %a 1
    if result >= 2 ^ (maxPowerOfTwo - (index - 1))
        if randomNumbers(index + modSchemeOrderOffset) == chromosomeLength
        else
            %Add 2 to index power to modulation order
            mutatedPower = mutatedPower + 2 ^ (maxPowerOfTwo - (index - 1));
        end
    else
        %Had a zero
        if randomNumbers(index + modSchemeOrderOffset) == chromosomeLength
            mutatedPower = mutatedPower + 2 ^ (maxPowerOfTwo - (index - 1));
        end
    end
end

%Mutate bandwidth
mutatedBandwidth = 1;
modSchemeOrderPowerOffset = 10;
bandwidthBits = 1:1:5;
for index = bandwidthBits
    %Determine if bit is a one or zero
    maxPowerOfTwo = 4;
    index2 = 1;
    result = recombinedChromosome(4) - 1;
    while index2 < index
        if result >= 2 ^ (maxPowerOfTwo - (index2 - 1))
            result = result - 2 ^ (maxPowerOfTwo - (index2 - 1));
        end
        index2 = index2 + 1;
    end

    %If result is greater than or equal to 2 raised to index power, had
    %a 1
    if result >= 2 ^ (maxPowerOfTwo - (index - 1))
        if randomNumbers(index + modSchemeOrderPowerOffset) == chromosomeLength
```

```
            else
                %Add 2 to index power to modulation order
                mutatedBandwidth = mutatedBandwidth + 2 ^ (maxPowerOfTwo - (index - 1));
            end
        else
            %Had a zero
            if randomNumbers(index + modSchemeOrderPowerOffset) == chromosomeLength
                mutatedBandwidth = mutatedBandwidth + 2 ^ (maxPowerOfTwo - (index - 1));
            end
        end
    end
end

%Recombines chromosomes; probabilistically picks one of the two genes from
%each chromosome for gene for new chromosome and returns
function [recombinedModScheme, recombinedModOrder, recombinedPower, recombinedBandwidth] =...
    recombineChromosomes(firstChromosome, secondChromosome)
    %Generate four random 0s or 1s
    %If the result for a given gene is a 0, take the gene from the first
    %population member
    %Otherwise, take the gene from the second population member
    randomNumbers = randi([0 1], 1, 4);

        %Modulation scheme
    if randomNumbers(1) == 0
        recombinedModScheme = firstChromosome(1);
    else
        recombinedModScheme = secondChromosome(1);
    end

    %Modulation order
    if randomNumbers(2) == 0
        recombinedModOrder = firstChromosome(2);
    else
        recombinedModOrder = secondChromosome(2);
    end

    %Modulation scheme
    if randomNumbers(3) == 0
        recombinedPower = firstChromosome(3);
    else
        recombinedPower = secondChromosome(3);
    end

    %Modulation scheme
    if randomNumbers(4) == 0
        recombinedBandwidth = firstChromosome(4);
    else
        recombinedBandwidth = secondChromosome(4);
    end

    %Check to ensure we are not combining QAM with an odd modulation order
    %If so, replace with even order
    evenModulationOrders = [2, 4, 6, 8];

    if(recombinedModScheme == 1 && mod(recombinedModOrder, 2) ~= 0)
        newOrder = randi([1 4], 1, 1);
        recombinedModOrder = evenModulationOrders(newOrder(1));
    end
end
```

# Appendix D. Frequency Bar Charts for Engine Parameters

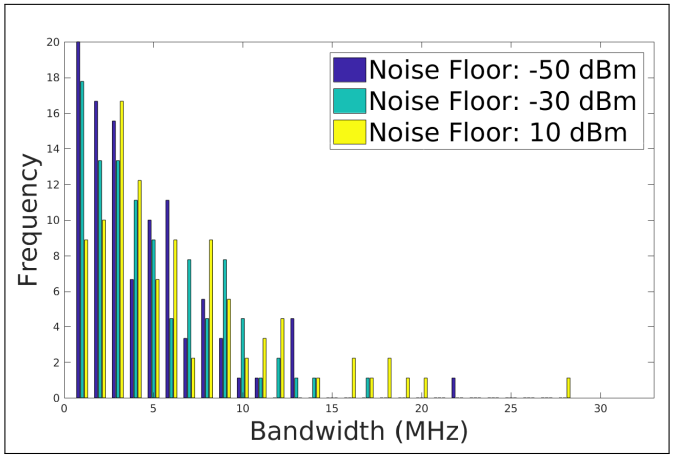


**Figure 36. Probability engine returns each possible bandwidth value for each noise floor when FEC in use**

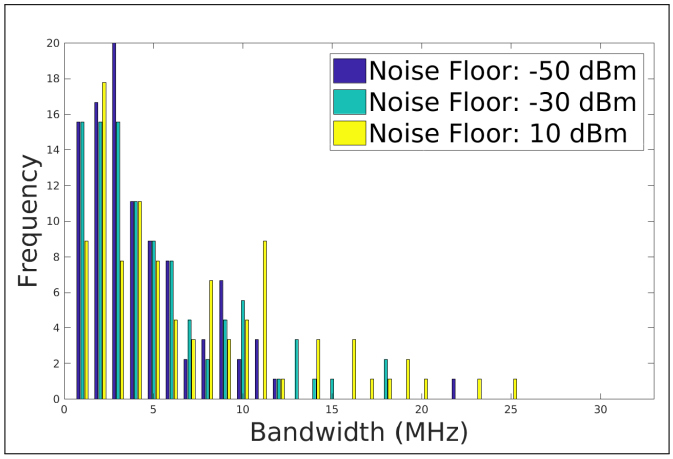


**Figure 37. Probability engine returns each possible bandwidth value for each noise floor when FEC not in use**

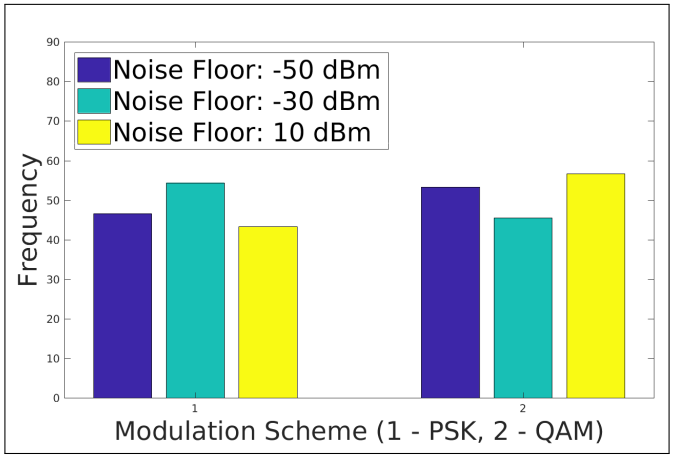


**Figure 38. Probability engine returns each modulation scheme for each noise floor when FEC in use**

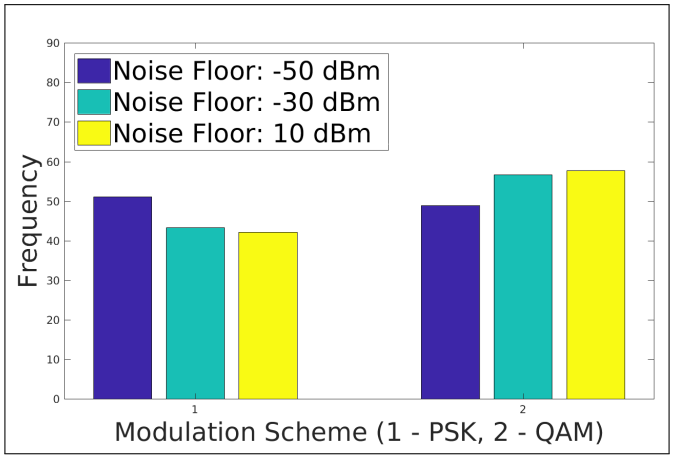


**Figure 39. Probability engine returns each modulation scheme for each noise floor when FEC not in use**

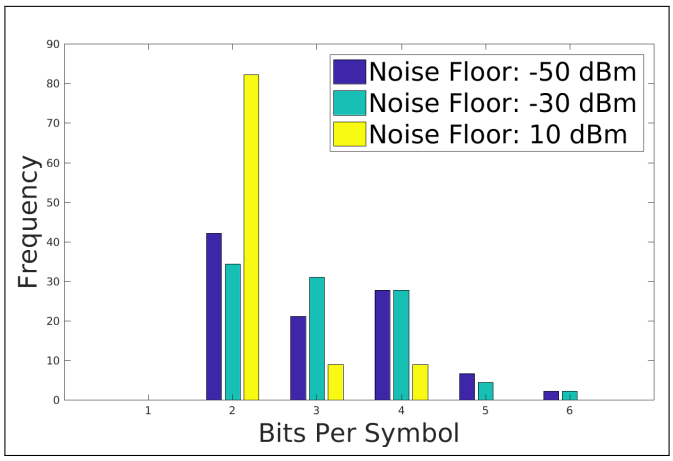


**Figure 40. Probability engine returns each possible bits per symbol value for each noise floor when FEC in use**

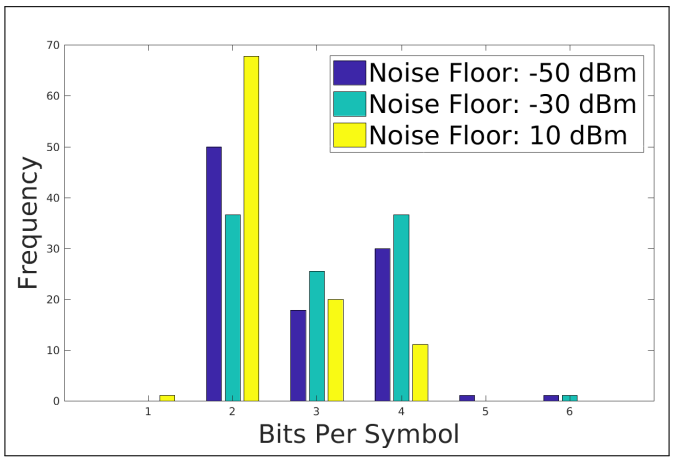


**Figure 41. Probability engine returns each possible bits per symbol value for each noise floor when FEC not in use**

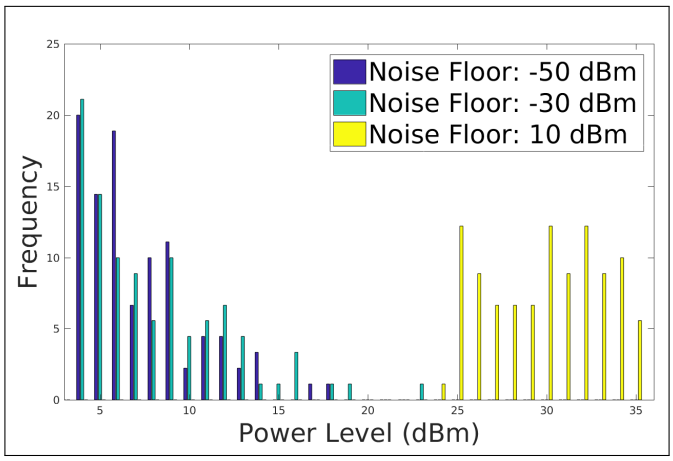


**Figure 42. Probability engine returns each possible power value for each noise floor when FEC in use**

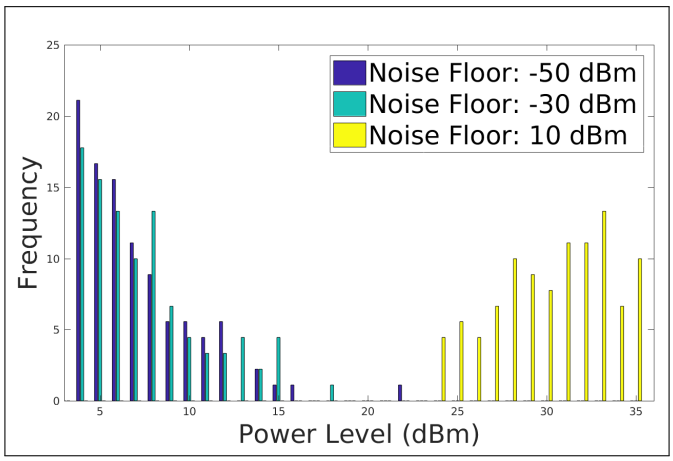


**Figure 43. Probability engine returns each possible power value for each noise floor when FEC not in use**

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704–0188*



| 1. REPORT DATE *(DD–MM–YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From — To)* |
|---|---|---|
| 22–03–2018 | Master's Thesis | Sept 2016 — Mar 2018 |

**4. TITLE AND SUBTITLE**

Techniques for Low-Latency in Software-Defined Radio-Based Networks

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**

Hart, Daniel D., Capt, USAF

**5d. PROJECT NUMBER**

18G334G

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

**8. PERFORMING ORGANIZATION REPORT NUMBER**

AFIT-ENG-MS-18-M-032

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

Air Force Research Laboratory - Space Vehicles Directorate
3550 Aberdeen SE
Kirtland AFB, NM 87117-5776
DSN 246-5812, COMM 505-846-5812
Email: james.lyke.2@us.af.mil

**10. SPONSOR/MONITOR'S ACRONYM(S)**

AFRL/RVS

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

**12. DISTRIBUTION / AVAILABILITY STATEMENT**

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**13. SUPPLEMENTARY NOTES**

This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States

**14. ABSTRACT**

Decreased budgets have pushed the United States Air Force towards using existing systems in new ways. The use of unmanned aerial vehicle swarms is one example of reuse of existing systems. One problem with the increased utilization of these swarms is the congestion of the electromagnetic spectrum. Software-defined or cognitive radios have been proposed as a basis for a potential robust communications solution. The present research aims to develop and test a genetic algorithm-based cognitive engine to begin looking at real-time engines that could be used in future swarms. Here, latency is the optimization objective of primary importance. In testing the engine, particular items of interest include the number of solutions evaluated in a given bound and the engine's reliability in yielding acceptable network performance. Initial experiments indicate the engine can consider significant portions of the search space within a relatively small bound and that the engine is efficient at finding highly fit solutions. Future work for this research includes evaluating how well high fitness correlates to acceptable performance and testing the engine with additional noise floors.

**15. SUBJECT TERMS**

Bandwidth, Cognitive Engine, Cognitive Networking, Cognitive Radio, Genetic Algorithm, Latency, Phase Shift Keying, Quadrature Amplitude Modulation, Software-Defined Radio, Throughput, Transmission Power

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Dr. Kenneth M. Hopkinson, AFIT/ENG |
| U | U | U | U | 205 | 19b. TELEPHONE NUMBER *(include area code)* (937)255-3636 x4579; Kenneth.Hopkinson@afit.edu |