SIMULATION OF AN ANTI-COLLISION ALGORITHM FOR RFID SYSTEMS USING A CODE DIVISION MULTIPLE ACCESS WITH ADAPTIVE INTERFERENCE CANCELLATION (CDMA/AIC) APPROACH WITH DYNAMIC PROCESSING GAIN (GP)

by

Natalia Margarita Benitez Gutierrez, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Electrical Engineering
December 2022

Committee Members:

Harold Stern, Chair

William Stapleton

Cecil R. Compeau

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

This thesis is dedicated to my parents, who supported me all this time with their love and prayers, even being far.

To my sister, who is my inspiration every day.

To my boyfriend, for all the love and support.

To my friends and my professors, for all the trust and best wishes.

ACKNOWLEDGEMENTS

Thank you to my thesis advisor, **Dr. Harold Stern**. I am grateful for all the teaching, time, patience, and advice he gave me during my thesis work and as his student and TA. It was an enriching experience to work beside an outstanding professional and an excellent person. His motivation, trust, and wisdom successfully inspired me to create this shared achievement with him.

I am also grateful to **Dr. Stapleton** and **Dr. Compeau** for the time and valuable advice they gave me for this thesis work.

I am incredibly grateful to my father, **Diego Benitez**, and my mother, **Silvia Gutierrez**, for being my example of love and effort and supporting my dreams. It was not easy being so far away, but we made it! Thank you to my younger sister **Daniela Benitez**, who supported me with love and blessings.

Thank you to my boyfriend, **John Frerich**, for his love and advice during this work.

Finally, I'd like to thank all my professors and friends who contributed to my education and made me the person I'm today.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS ……... v

LIST OF TABLES..……...……..…… viii

LIST OF FIGURES …….. ix

LIST OF ABBREVIATIONS …… x

ABSTRACT …….…… xi

CHAPTER

I. INTRODUCTION …..…….....1

II. PREVIOUS RESEARCH ……...……..... 3

A. TDMA-based systems ..……..... 3
B. CDMA-based systems ..……..... 7

III. NEW RESEARCH: CDMA/AIC WITH DYNAMIC PROCESSING GAIN……..……..... 14

IV. DEVELOPMENT OF CODE FOR CDMA/AIC WITH DYNAMIC PROCESSING GAIN …...……..... 18

A. Explanation of the code ……..... 20
B. Flow diagram ……..... 22

V. TESTS …...……..... 24

A. Repeatability tests ……..... 24
Code division multiple access (CDMA) 25
Dynamic framed slotted ALOHA (DFSA) 26
B. Time tests ……..... 27

VI. ANALYSIS ……..... 30

A. First test 30
B. Second test 31
C. Third test 32
D. Fourth test 33

VII. CONCLUSIONS 36

VIII. FUTURE RESEARCH SUGGESTIONS 39

APPENDIX SECTION 41

REFERENCES 88

LIST OF TABLES

Table **Page**

1. Different anti-collision algorithms with ALOHA Protocol [3,4,7] ……... 4

2. Distinctions between previous research work and CDMA/AIC with Dynamic Processing Gain ……... 15

3. Repeatability tests with 60 tags, Gp=64 and 256 bits ……. 26

4. Repeatability tests with 60 tags, slots=64 and 256 bits ……... 27

5. First test with the CDMA and DFSA code, 1000 runs and 12 dB SNR ……. 28

6. Second test with the CDMA and DFSA code, 1000 runs and 9 dB SNR ……... 28

7. Third test with the CDMA and DFSA code, 1000 runs and 6 dB SNR ……..... 29

8. Fourth test with the CDMA code, 1000 runs, 3dB and 0 dB SNR ……. 29

9. Successful demodulation times for a CDMA/AIC system with dynamic processing gain, for different noise levels and the same initial Gp=128 ……..... 35

LIST OF FIGURES

Figure **Page**

1. Anti-collision Protocol classification [4] ……..…… 4

2. Q algorithm used in DFSA [5] …….. 6

3. Explanation of messages transmission with CDMA [10] …….....8

4. Amplitude of the tags modeled as a Rayleigh distribution ……..... 12

5. Time plot for CDMA/AIC (blue) and DFSA (orange) when exposed to the same conditions: 12 dB SNR, 128 Gp/slots, 1000 runs ……... 31

6. Time plot for CDMA/AIC (blue) and DFSA (orange) when exposed to the same conditions: 9 dB SNR, 128 Gp/slots, 1000 runs …….. 32

7. Time plot for CDMA/AIC with dynamic processing gain when exposed to 6 dB SNR, initial Gp = 128, 1000 runs …… 33

8. Time plot for CDMA/AIC when exposed to: 3 dB SNR and 0 dB SNR, initial Gp=128, 1000 runs. ……..... 34

9. Time performance of CDMA/AIC when exposed to different noise levels (12 dB, 9 dB, 6 dB SNR) and a Gp = 128 ……... 35

LIST OF ABBREVIATIONS

Abbreviation	Description
RFID	Radio Frequency Identification
CDMA	Code Division Multiple Access
TDMA	Time Division Multiple Access
SDMA	Space Division Multiple Access
FDMA	Frequency Division Multiple Access
SS	Spread Spectrum
AIC	Adaptive Interference Cancellation
DFSA	Dynamic Framed Slotted ALOHA
FSA	Framed Slotted ALOHA
PN	Pseudo-Random Codes
GP	Processing Gain
LF	Low Frequency
HF	High Frequency
UHF	Ultra High Frequency
BW	Bandwidth
ALOHA	Additive Links On-line Hawaii Area
ACK / NACK	Acknowledgement / No Acknowledgement
MPR	Multi-Packet Reception

ABSTRACT

Radio Frequency Identification (RFID) Systems are modern wireless communication systems that transmit information from a transponder (tag) to a reader. RFID systems are well known because of their contactless feature. However, tag performance is limited by collision problems when multiple tags transmit simultaneously. Due to the collision problem, much research has been developed using anti-collision algorithms to enhance the systems' efficiency, save energy, and ensure the correct transmission of information. Most research has used a Time Division Multiple Access (TDMA) approach with anti-collision ALOHA-type algorithms. The time slots and frames of the tags are manipulated to deal with the collision problem. They work with different ALOHA protocol variants that are always trying to reduce the number of collisions compared to the previous techniques. The most promising of the ALOHA protocol variants is Dynamic Frame Slotted ALOHA (DFSA). In addition, research has been conducted with a Code Division Multiple Access (CDMA) approach, called CDMA with Adaptive Interference Cancellation (CDMA/AIC). The time slots are not used for this anti-collision algorithm; instead, Spread Spectrum (SS) technology and Processing Gain (Gp) were employed. In previous work, the Gp was a fixed value equal to sixty-four (64). In contrast, this research involved a CDMA/AIC approach with a dynamic Gp reached by generating different chip rates. This technique depended on the number of collisions from the previous run to resize the Gp for a subsequent run. CDMA gave the flexibility to use Spread Spectrum. The modulated signal was spread across the channel

using orthogonal pseudorandom (PN) codes (generated for each tag) and was demodulated at the reader using the same PN code. The more spread the signal was in the channel, the greater the Gp. The research proved an enhancement in the system's performance compared to the previous work. The system's efficiency enhancement and the anti-collision algorithm were proven using MatLab as the simulation software. No hardware implementation was developed in this research. Both the CDMA and the modified DFSA code were exposed to the same conditions of noise (12, 9, 6 dB SNR), number of tags (20, 60, 80, 100, 150, and 200), number of simulations (1000), and Gp/slots (32, 64, 128, and 256). After the data was collected and processed, the performance of CDMA in noisy scenarios and with a large number of tags was faster and more efficient than DFSA. CDMA presented stability and fast information processing due to its error correction and code spreading features.

I. INTRODUCTION

RFID is an acronym for Radio Frequency Identification. RFID systems use wireless communication techniques in order to transmit and receive data from the transponders to the readers. The transponders are the devices that contain the data; they are also called tags [1]. The information exchange is made with radio waves using transmitting and receiving antennas. Depending on the frequency band they are working in, the RFID systems can be classified as Low-Frequency systems (LF), High-Frequency systems (HF), Ultra High-Frequency systems (UHF), or Microwave systems [2].

RFID systems evolve with time, and to improve performance, the systems' developers need to find ways of avoiding collisions in the transmission process. In some processes, information loss is not an option, so the interpretation of tag information must be precise. When an RFID system tries to read the tags within the reading range, collisions may occur due to the number of tags within the area. When collisions occur, the information is lost, and a retransmission of the collided tags is needed. This retransmission introduces a delay in the transmission, and the performance of the system decreases.

Although RFID systems are widely used in industry due to the contactless transmission feature, they have significant limitations because the data coming from the tags are not always clearly read by the reader. The collisions that occur during the transmission decrease the system's efficiency. Some methods have been applied to RFID systems to solve this problem, including the enhancement of the Bandwidth (BW) in the communication channel for better transmission speed [3].

Many models of anti-collision algorithms have been proposed, primarily

involving Time Division Multiple Access (TDMA) and ALOHA protocol criteria. Pure ALOHA, Slotted ALOHA, and framed slotted ALOHA are the three main protocols used up until now. Some studies and some systems have applied Dynamic Framed Slotted ALOHA (DFSA), and when this protocol is used, the system performs better due to its dynamic feature. The dynamization of the frame size (and the number of slots in a frame) lets the anti-collision algorithm test the number of collisions. If the number of collisions is high, the frame size needs to be increased. On the other hand, few to no collisions mean that the frame size can be maintained or decreased. The size of the frame adapts dynamically, and it always depends on the number of collisions that exist in the previous run [3].

This research work focuses its effort on developing an algorithm similar to DFSA anti-collision algorithms. However, instead of dynamic framed slotted ALOHA, the system will use Code Division Multiple Access (CDMA) and dynamically vary processing gain (Gp). This approach makes the system suitable for RFID transmissions where collisions are very likely to occur, and background noise may be high. The dynamization of the Gp depends on the dynamic pseudo-random codes that will be generated with a Code Division Multiple Access (CDMA) modulation technique. The dynamization of the Gp will be similar to the process used to generate a dynamic frame size for the ALOHA protocol for the transmitted data and will guarantee an enhancement in the transmission, which means that the system will be efficient enough to avoid collisions that were likely to appear in non-dynamic anti-collision techniques.

II. PREVIOUS RESEARCH

RFID systems are well known for their feature of having a large number of tags that send their information wirelessly with or without having a direct line-of-sight (between tag and reader) [4]. This feature creates a significant problem which is the tag collision.

A. TDMA-based systems

There are two types of TDMA-based anti-collision algorithms that have been developed and tested: binary-tree-based algorithms and Aloha-protocol-based algorithms. The binary-tree-based algorithms are not commonly used due to their slow processing when there is a large number of tags in a system. In other words, it is useless when large-scale real-time applications are required. On the other hand, ALOHA-type algorithms are widely used because of their simplicity, flexibility, and low cost [5]. ALOHA-type anti-collision algorithms have been developed in order to reduce or ease the collision problem and enhance the performance of the systems in their different applications in industry, eluding excessive bandwidth and energy waste (during a collision) [4]. Figure 1 shows the different TDMA-based anti-collision protocols used in RFID systems and variants.

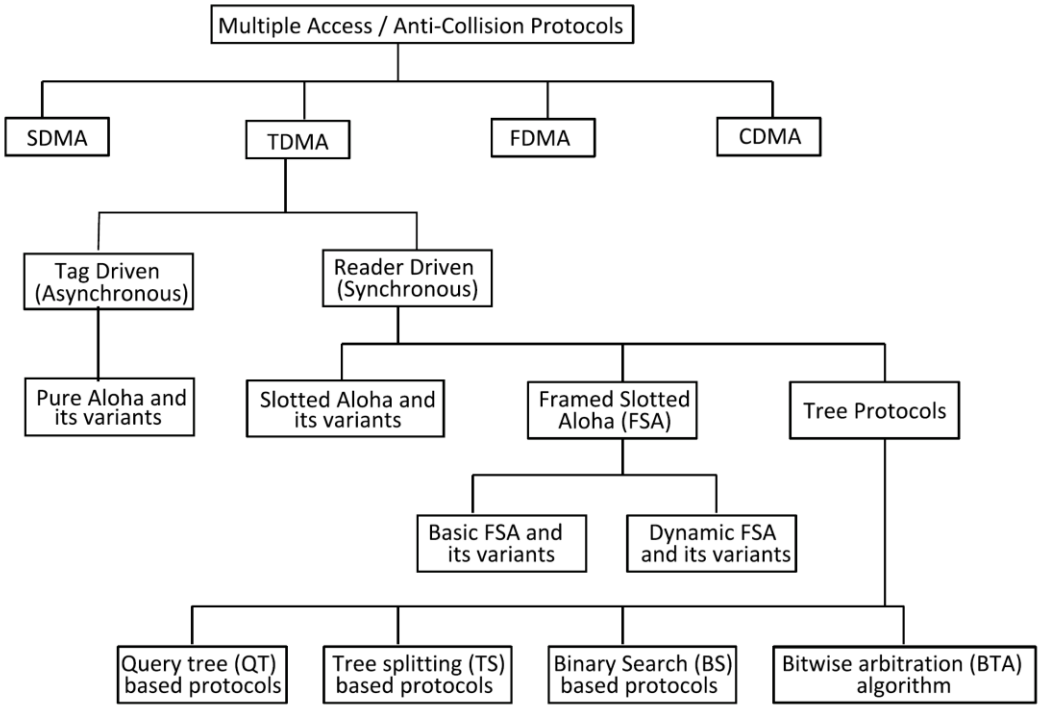


Figure 1. Anti-collision Protocol classification [4].

The research works using TDMA-based anti-collision algorithms were conducted using different ALOHA protocols: Pure ALOHA, Slotted ALOHA, Framed Slotted ALOHA, Dynamic Framed Slotted ALOHA, and its variants [3,4,7]. Table 1 below describes the overall idea of each anti-collision algorithm with ALOHA protocol.

Table 1. Different anti-collision algorithms with ALOHA Protocol [3,4,7]

Pure ALOHA	It is a TDMA anti-collision protocol; it is very basic compared to other ALOHA variants. When the tags are in the interrogation zone, they send their ID to the reader with a random transmission frequency within the communication channel. The reader answers with an acknowledgment (ACK) or a no acknowledgment (NACK). When NACK occurs, it means that a collision happened in the system. Collisions happen due to multiple tags within the interrogation zone transmitting simultaneously. When collisions occur, a retransmission of the collided tags is needed **[4, 7]**. A NACK may also occur due to thermal noise, causing errors when the tag is read.

Slotted ALOHA	It uses time slots or intervals. For the transmission, each tag randomly selects a time slot for transmitting its information (synchronized communication). A tag is identified when it occupies a time slot. The condition to avoid collisions is that the time slot will be occupied by one and only one tag. When collisions occur, or thermal noise causes errors, the tags retransmit their information after a random amount of time [3, 4, 7].
Framed Slotted ALOHA	It uses frames of a fixed size divided into slots of time. The fixed frames are used for the transmission process. The FSA protocol provides the reader with information regarding the frame size and the random numbers assigned to the tags to select the slot into a frame. The tag interchanges its information in a fixed frame size; each tag will transmit its information only in one slot of the frame, which means that each tag will only send information once in the whole frame. The main problem with FSA is that if the frame size is too small and the number of tags is large, many (or even all) of the slots may have collisions. Alternatively, the other scenario occurs when the frame size is too large, and there is inefficiency and a significant delay introduced to the system due to large-sized frames used for the small number of tags [3, 7].
Dynamic Framed Slotted ALOHA	It is similar to FSA but with dynamic frame sizes. Per reading round, the reader estimates the tags' information and, depending on it, modifies the size of the frame. This calculation inside the reader is called an "estimation function". It is determined depending on the number of tags the system has, the number of used slots within a frame, and the number of slots with collisions. The main problem with DFSA is that it requires high computational calculations for the frame estimation, depending on the method used [3, 4, 7]. DFSA has 2 popular methods to assign the size of the fame, they are: • Algorithms that use the number of slots that are empty, collided slots, and the slots with only one tag [3]. • Algorithms that begin a cycle of frame reading with two or four slots, and if no tag is correctly read in that frame, the size is increased in the next reading cycle [3]. DFSA research has proposed some methods of analysis for the frame size and has some variations to find an optimal system with the fewest collisions for a particular application. They include: • Advanced Framed Slotted ALOHA • Dynamic Framed Slotted ALOHA • DFSA with Query (Q) algorithm • DFSA with binomial analysis • FSA-MPR

ISO18000-6C is the communication standard for passive RFID systems that use Ultra High Frequency (UHF) Class 1 Generation 2 tags. This standard defines the

codification of the information in the tags, the modulation technique used, and the anti-collision protocols that will be used [6]. Based on this standard, the Query (Q) algorithm (which is a form of Slotted ALOHA) is used in some research works with Dynamic Framed Slotted Aloha (DFSA) in order to adjust the size of the frames dynamically, and it is done frame by frame [5]. Figure 2 summarizes the process of the Q algorithm during the communication between the interrogator (reader) and the transponder (tag). DFSA research has proposed some methods of analysis for the frame size and has some variations to find an optimal system with the fewest collisions for a particular application.

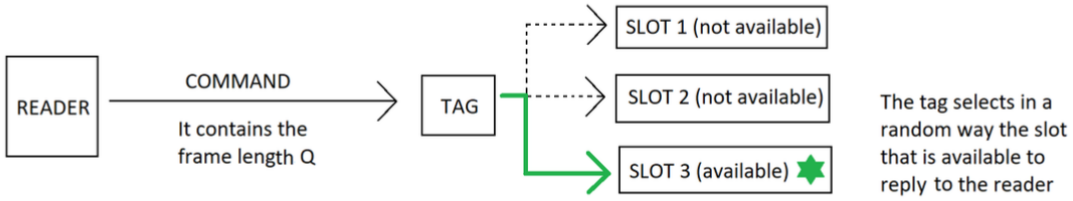


Figure 2. Q algorithm used in DFSA [5].

When using the Q-algorithm, Zuliang et al. note that variable C (a calculation parameter for resizing the Q value or the frame) is used to increment or decrement the Q value, providing a dynamic frame size adjustment. The value of C ideally varies from 0.1 to 0.5. But in real applications, it can be higher or lower. If the value of C is too high, the length of the frame needs to be adjusted frequently, and the system becomes unstable. On the contrary, if the value of C is too low, the frame adjustment is slow [5]. An equilibrium is needed for the frame sizing purposes because both scenarios are undesired. Q (changing its values due to C) being too large with a small number of tags introduces delays, and Q being too small with a large number of tags increases the probability of

collisions [13]. The researchers compared the Q-algorithm with the DFSA and developed an enhancement in the global output of the system. They suggest that there should be a balance between delays and identification precision when using a large number of tags, and grouping tags is an option for analysis with a large number of tags [5].

Zuliang et al. use a formula for the Q algorithm to estimate the initial tag population. Q is the length of the frame, and n is the number of tags they identify. The probability that r number of tags will select a slot at the same time is calculated with **[5]**:

$$P(Q,n,r) = C_n^r * \left(\frac{1}{Q}\right)^r * \left(1-\frac{1}{Q}\right)^{n-r}$$

Due to the case that r = 0, 1, or greater than 1, the probability of an empty slot, a successfully filled slot, and collision in slot can be calculated with the equation. For the example, for the case that r = 1 [5]:

$$P(Q,n,1) = \frac{n}{Q} * \left(1-\frac{1}{Q}\right)^{n-1}$$

B. CDMA-based systems

Code division multiple access or CDMA is a modulation technique in which several users can transmit their data simultaneously using the same frequency channel. CDMA uses pseudo-random codes that are orthogonal between each other to spread the messages among the channel and take advantage of the spread spectrum. The characteristic of pseudo-random codes, as mentioned before, is that they must be orthogonal between each other [10, 11]. The pseudo-random codes play an essential role in CDMA because the transmitter modulates the signal using these codes. At the receiver, the pseudo-random codes must be known to demodulate the received signal. The

modulation and the demodulation of the transmitted signals are done using XOR functions, as shown in Figure 3.

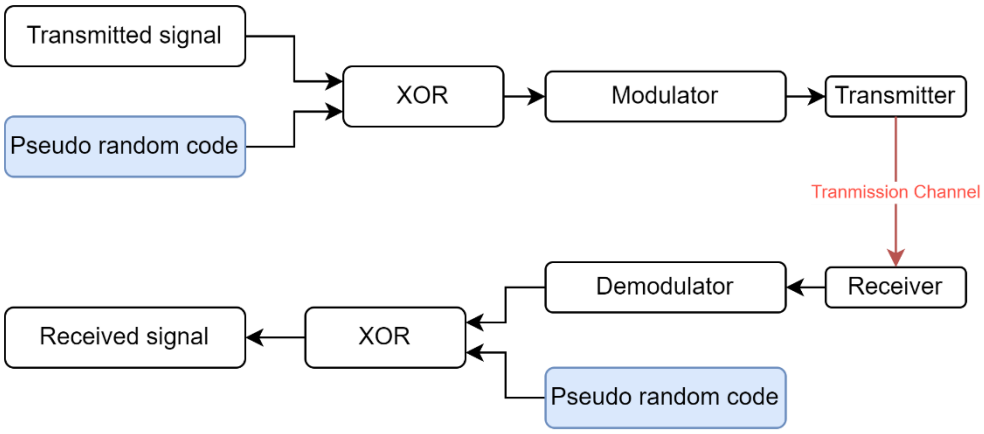


Figure 3. Explanation of messages transmission with CDMA [10].

When working with CDMA, the spread spectrum is an important characteristic. The spread spectrum in communication systems has the following characteristics:

- The transmitted signal has a bandwidth bigger than the minimum bandwidth required for the transmission [10].
- The pseudorandom codes are not related to the transmitted information, but they are the ones that determine the chip rate and the size of the bandwidth. As long as the pseudorandom codes are longer than the information messages, the bandwidth will be increased in size [10].
- At the receiver, all the pseudorandom codes of the transmitted signals are known, so demodulation is possible, and the original signal is received **[10]**.

The term "Processing Gain" (Gp) appears when working with CDMA systems.

This term is not used in TDMA anti-collision algorithms. The difference between TDMA and CDMA systems is that spreading codes in CDMA makes the system more flexible for dynamic change. In this particular case, it is a type of dynamic collision control but with variable Gp instead of variable frame size. Using CDMA as a modulation technique lets various tags use the same channel at the same time once their information is spread out over the communication channel. The CDMA process provides extra protection against thermal noise and enhances the capability to demodulate weak signals successfully.

Although most anti-collision research for RFID systems has involved TDMA, some researchers focused on creating anti-collision algorithms using a CDMA modulation technique. Vahedi et al. model their CDMA system using a two-dimensional absorbing Markov chain. This technique helps identify the tags in the CDMA system so the results in the number of transmitted bits and demodulation time can be compared with EPC Gen 2 standard, which suggests using Dynamic Framed slotted ALOHA with a Q-algorithm. The Markov chain absorbing states used by the researchers are single, idle, and collided transmission. From this model, they use the following criteria to move through the state's map: if an idle transmission exists, the number of tags does not change, but the Q value decreases (changing the C value), and they jump to the next state. If there is a collided transmission, the number of tags does not change, but the Q value increases (changing the C value), and they jump to the next state [13].

Maina et al. propose an application for checking out merchandise in a store. Since CDMA modulation allows simultaneous transmission, the information of the tags can be transmitted and processed simultaneously. The application also uses multiple reading

devices, and adjacent interference, as one of their main problems, was managed through 2 techniques: clear distance technique and directed signal technique. For the first method, the energy of each symbol is calculated as shown in the equation below [14]:

$$E_s = \frac{E_c}{n}$$

where: E_s is the Energy of the symbol, E_c is the energy per chip, and n is the number of tags involved in the process. Examining the configuration of the store, as long as the distance between the aisles is a main parameter of concern, they calculate the acceptable distance with the inverse square law, and the results of these calculations ended up with a distance between aisles ≥ 8 feet (to avoid undesired interferences). For the case of the directed signal technique, they considered that not all the tags would be aligned. They solved the problem by synchronizing the readers so that just one is reading at the time, and the next reads in the next clock cycle, and so on. Also, one reader is an array of 3 devices, one in the front of the cart and two on the sides, a collaborative reading is used to cover most of the antenna directions [14].

Wuu et al., in their research, propose a Zero-collision RFID tag identification system using CDMA modulation and hash-chain technology [15]. The hash chain technology is used for security purposes; several levels of security are applied because the characteristic of a hash chain is producing one-time keys from a single key, so a chain of length three is expressed as $h^3(x) = h(h(h(x)))$ [16].

Previous research has been conducted using a Direct Sequence CDMA/AIC anti-collision algorithm with fixed processing gain with a value of 64 [8]. CDMA/AIC stands for Code Division Multiple Access with Adaptive Interference Cancellation. Its objective is to control better a system exposed to multipath problems, shadowing, and the near-far

problem. This algorithm uses the signals received from the reader, the information from the tag, the interference from the other tags within the interrogation zone, and the noise from the environment. With the AIC, the tag with the strongest signal is decoded first and taken out of the system. The tag with the second strongest signal is then decoded, taken out of the system, etc. The process continues until no additional tags can be successfully decoded. In the next run, decoded tags are not going to transmit. They will not cause interference, so fewer tags are transmitted in the system, the probability of collisions is reduced, and the energy is saved. With the CDMA-AIC anti-collision algorithm, the system's energy consumption is reduced because, unlike slotted ALOHA systems, the system will not generate slots that are not used [8]. The CDMA/AIC system with fixed Gp compared favorably with a Framed Slotted ALOHA system having fixed frame size [8], albeit using a small number of tags (8 – 12 tag capacity).

It is well known that problems like multipath, shadowing, backscattering, and near-far problems are likely to appear in wireless communication systems. One way to reduce their effect on the communication channel is by employing Adaptive Interference Cancellation (AIC) techniques. Once the signal is transmitted from the tag to the reader, it is not that pure signal that arrives back at the reader. Instead, the signal will have multiple interference components from other tags and the environment. Therefore, in a natural environment, the signal that arrives to the reader will have components of interference and noise [8, 17].

The AIC technique in this thesis follows the same parameters as the reference [8], using a Rayleigh fading distribution which models a multipath propagation environment with no line-of-sight components. During the transmission, the amplitude of each’s signa

is computed using a Rayleigh probability density function with a scale factor of 1, as shown in Figure 4. Depending on the variation of power among the transmitted tags, the tag with the strongest power level is demodulated first and its effects are removed from the system. The tag with the second strongest power level is demodulated right after the first one, its effects are removed from the system, and so on. This process is done until no more tags can be successfully demodulated.

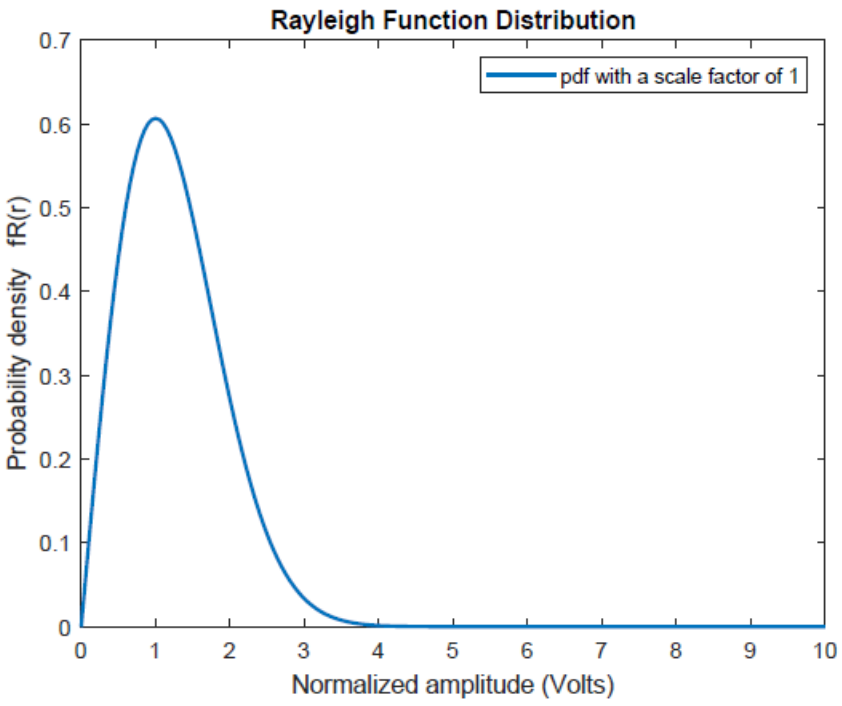


Figure 4. Amplitude of the tags modeled as a Rayleigh distribution.

The tags that were successfully demodulated are removed from the system, so on the next run, only the tags that were not demodulated will be part of the process. The same approach is used in the next run, the power between the tags is compared, and the most powerful is demodulated first. If the power of some tags in the system is very low,

they will be part of the next run, and their power will be different so that they will be successfully demodulated.

III. NEW RESEARCH: CDMA/AIC WITH DYNAMIC PROCESSING GAIN

The research in this thesis will use CDMA/AIC (as proposed in reference [8]) but with an adaptive Gp. Reference [13] used an adaptive Gp but did not employ AIC. The algorithm to adjust Gp will be less complex than the one proposed in reference [13]. The dynamic Gp will generate different chip rates depending on the number of collisions from a previous run. As with Reference [8], the research in this thesis will be conducted in a Rayleigh fading environment. As it is known, in an RFID wireless communication system, shadowing is very likely to happen. Within a bulk of tags transmitting simultaneously, multipath will occur and shadowing will cause some tags to not have a direct line of sight to the reader. In such a process, a Rayleigh distribution is the most suitable model for the RF propagation.

Reference [12] examines ways to evaluate CDMA/AIC for other environments modeled with different distributions like Rician and Lognormal in order to enhance parameters like extending the line of sight in the RFID system. It is essential to highlight that in both [8] and [12], the processing gain for the simulations was fixed and with a value of 64. The difference between previous DFSA research work and the work in this thesis is that it is not a TDMA system anymore. The system is purely CDMA, as shown in Figure 1 (upper right). Additionally, opposite to references [8] and [12], the Gp is dynamic and changes automatically. In this work, it is possible to operate with significantly more than 8-12 tags simultaneously, which was the capacity of the reference [8] work. Table 2 below shows the distinctions between the previous research and CDMA/AIC with Dynamic Processing Gain.

Table 2. Distinctions between previous research work and CDMA/AIC with Dynamic Processing Gain

Reference	TDMA or CDMA-based?	Dynamically adjusted slot size or Gp?	Rayleigh fading environment?	Includes effects of noise?
[3]	TDMA	Yes	No	No
[5]	TDMA	Yes	No	No
[7]	TDMA	Yes	No	No
[8]	CDMA	No	Yes	Yes
[9]	CDMA	No	Yes	Yes
[13]	CDMA	Yes	No	No
[14]	CDMA	No	No	Yes
[15]	CDMA	No	No	No
New CDMA/AIC with Dynamic Gp	CDMA	Yes	Yes	Yes

Depending on the number of collisions, the simulated system can dynamically generate orthogonal pseudo-random codes, which will change the value of the Gp. The dynamization of Gp will cause the number of collisions to be reduced in the system. The more collisions exist, the greater the Gp will be before the next run, and vice versa.

In CDMA-based systems, the spread spectrum is the main concept. This is because there can be many users using the same channel at the same time. This is achieved when the transmitted signals are spread out on the channel depending on the spreading code [9]. For our purpose, the analysis of the Spread Spectrum (SS) has an important role. The greater the Gp (i.e., the more SS in the channel), the fewer unsuccessfully demodulated tags would be expected, but if Gp is too large, bandwidth is wasted, and effective transmission speed is reduced. So, there should be a balance between SS and the spreading codes (Gp) length to maintain the system's performance.

The analysis will be made in such a way that Gp will be large enough for all the

tags to transmit simultaneously. Suppose the spread message has a dynamic size depending on the information of all the tags transmitting simultaneously. In that case, fewer resources will be wasted in the system, and its efficiency will increase because we can consider a parallel transmission of the information. It is important to highlight that the change in the size of the Gp is independent of the size of the information of a single tag. It depends on the number of tags that are transmitting at the same time and using the same communication channel.

MATLAB has been used for the simulation, with programs developed based on a previous work where CDMA/AIC was used with fixed Gp. The programs will show the number of collisions, the runs, and the Gp per run. As described below, programs will also be developed to evaluate the performance of DFSA systems.

The "Tests" section of this document will show the comparison of 2 systems working under similar conditions and the outputs they generate in order to determine the effectiveness of the CDMA code. The codes that are part of the test process will be the CDMA/AIC code with dynamic processing gain and an FSA (framed slotted ALOHA) code extensively modified from a previous source to allow dynamic changes in frame size [8]. Changes to the FSA code in order to work dynamically are:

- The user can input the initial parameters for the program, which are: the number of tags, number of slots, number of times that the program will run, and the message length. The amount of background noise is also adjustable in the code.
- The main code and three functions are created. The first function keeps the number of slots from the previous run, the second function doubles the number of slots from the previous run, and the last function reduces by half the number of

slots from the previous run. The main program will call the functions depending on the number of unsuccessfully demodulated tags from the previous run.

- Matrices with values of the variables (mentioned in the first bullet) and their changes during each run are created. A final matrix that superimposes all the small matrices is created at the end for evaluation.

IV. DEVELOPMENT OF CODE FOR CDMA/AIC WITH DYNAMIC PROCESSING GAIN

The code developed in a previous thesis work [8] was extensively modified to the researcher's needs, both for optimization and, more importantly, to incorporate automatic, dynamic adjustments of Gp. The main changes to the program are:

1. The new code has the feature of letting the user input the values for the process, and one of its values is the processing gain for the first run. The processing gain for the first run will be determined by the user and is not fixed as the original code in [8] (which was 64), and the processing gain will be automatically, dynamically adjusted in subsequent runs. The variables that the user inputs at the beginning are:

    a. Number of tags → In the original code, the number of tags was hard-coded and the system's performance with fixed Gp was such that the maximum practical number of tags was 12. With dynamic Gp the number of tags can be significantly larger than fixed Gp – the next portion of this thesis shows successful operation with up to 200 tags. Additionally, the number of tags can be greater than the Gp, that is a difference compared to the previous code in which the number of tags must be less or equal to the value of the Gp.
    b. Number of bits → Due to the work done under a Generation 2 Class 1 RFID system, there is a maximum of 256 information bits for each tag.

c. Number of simulations runs → Any positive number greater than zero is allowed. As will be shown later in the thesis, runs of 1000 simulations were sufficient to show stability of results.

d. Processing Gain → It's value should be a power of 2.

2. When generating the Hadamard matrix for creating the spreading codes, the original code transposed the matrix. The transposition of the matrix was not needed, some tests were made with the Hadamard values and its transposed matrix. For the cases of the power of 2 (as matrix size), they ended up being the same, so that piece of code was taken out.
3. To compare the demodulated signal with the original signal, in [8] a subtraction between these 2 arrays was made, if the result was 0 there was no error, else if 1, there was an error. In order to make the code more reliable, it was changed, a bit-by-bit comparator was created between these 2 arrays, and the same logic is used but with flags, if the result was 0 there was no error, else if 1, there was an error.
4. In the new program the percentage of successfully demodulated tags for the last run is computed and used to determine the new value of Gp. The Gp is dynamically adjusted in an automatic way.
5. Matrices of data are created for diagnoses, calculation, and test purposes. Individual matrixes with the data per run are combined into a big matrix with the data of all the process. The data in each matrix shows: the number of the

run, the number of rounds, the processing gain, the number of unsuccessfully demodulated tags, and the number of successfully demodulated tags.

6. The data from the consolidated matrix is exported to an Excel spreadsheet. From the spreadsheet the calculations for the tests are made.

A. Explanation of the code

The new program references the original code in [8]. An overall view of the code is that the original code was modified to be called from the new code as functions with the same Gp, half Gp, and double Gp. The previously mentioned output values of the codes are the percentage of successfully demodulated tags and the number of unsuccessfully demodulated tags. These outputs will determine if the processing gain needs to be changed in the next run. The code runs once and again while there are unsuccessful demodulated tags; it stops once the value of unsuccessfully demodulated tags is zero.

The explanation of the code is detailed below in 19 steps and a copy of the MATLAB code is provided in Appendix B.

1. Input the values of:
    a. Number of tags for the system
    b. Size of the information within the tags (number of bits).
    c. Number of times that the system will run.
    d. Initial processing gain for the first run.
2. Generate orthogonal spreading codes.
3. Assign the spreading codes to the tags.
4. XOR the information inside each tag with its respective spreading code.

5. Select a signal strength based on Rayleigh distribution.
6. Simultaneously transmit tags and add channel noise.
7. At receiver, check for the strongest signal in order to use AIC technique
8. Demodulate the information of the 1st strongest tag, mathematically remove its effects from the received signal, then demodulate the 2nd strongest tag, remove its effects, and so on at the receiver with the spreading codes
9. Compare the transmitter signals with the received signals
10. Determine the percentage of messages that were successfully demodulated.
11. Determine the number of messages in the collision that could not be successfully demodulated (integer).
12. Resize Gp depending on 3 possible scenarios:
    a. Double the processing gain → the percentage of successfully demodulated tags is between 0% and 40%.
    b. Keep the same processing gain → the percentage of successfully demodulated tags is between 41% and 70%.
    c. Reduce to the half the processing gain → the percentage of successfully demodulated tags is between 71% and 99%.

    The values of the thresholds can be adjusted within the program.
13. Retransmit using the same approach and the new Gp.
14. If there are still collisions, the code automatically repeats steps 10, 11 and 12 until all tags have transmitted successfully.
15. When all tags have been transmitted successfully, the program starts the next run.
16. After the las run, the elapsed time of the process is shown.

17. Plot a graphic to show the processing gain changes during the demodulation process.
18. Generate data matrices.
19. Export data to an Excel spreadsheet.

B. Flow diagram

The flow diagram of the code is shown in Appendix A. It explains how the CDMA/AIC program works with dynamic Gp and the variables used in the process. Appendix A also shows the variables map. In this map, there is a description of what the variables and the functions in the flow diagram do.

A brief explanation of the flow diagram follows: The program starts cleaning up any previous data, and it asks the user to input the values of the number of tags, number of bits, number of simulations, and the initial processing gain. The program runs for the first time and gets the number of successfully demodulated tags. If that value is between 0 and 40%, the Gp is doubled for the next run. If the successfully demodulated tags value is between 41% and 70%, the Gp value is the same for the next run. If the successfully demodulated tags value is between 71% and 99%, the Gp is reduced to half for the next run. If the successfully demodulated tags value is 100%, all the tags were successfully demodulated, a matrix of data is generated, and a plot of the Gp changes and successful demodulations is shown. If the simulation at that time is not the final run, the process starts all over again with the initial parameters. If the simulation is the final run, a matrix with all the process data is generated. The data is exported to Excel for further calculations.

The thresholds for doubling and halving Gp are adjustable within the program, and optimal values will depend on the number of tags and the amount of background noise in the system. Determination of optimal thresholds for various cases is beyond the scope of this thesis but is suggested as future research.

V. TESTS

Once the CDMA/AIC program with dynamic Gp was completed, it is necessary to run several tests in order to determine the system's efficiency compared to DFSA. These tests include the following:

- Repeatability test
- Time tests with different parameters mentioned below.
    - Different number of tags
    - Different number of Gp/slots
    - Different value of noise

In the time tests section, it will be shown how the time value is related to the key system performance parameters.

A. Repeatability tests

Repeatability in the implemented codes is essential and is the parameter that determines how many runs are necessary in order to produce stable results. The repeatability test was made with 60 tags, 256 bits, a processing gain Gp=64, and the number of simulations was varied from 50 to 2000 times. The most stable results were generated from simulation runs of 1000 or greater, where the fluctuation of the data was no more than 10% among five independent sets of samples. The data used to evaluate repeatability was a measurement of the amount of transmission time required for all the tags to be successfully demodulated.

For sampling purposes, both codes, CDMA and DFSA, were run for repeatability using background noise of 1.577 mV^2 which corresponds to 12 dB SNR. For calculating

the transmission time for successful demodulation of all tags, the following formula is applied:

$$t(sec) = \frac{\sum Gp}{n} * 256 * 40000$$

Where:

- $\sum Gp$ is the summation of all the processing gains during the run. (Note that for DFSA, this parameter will be replaced with the summation of all slots during the run.)
- n is the number of runs.
- 256 corresponds to the number of information bits that Class 1 Gen 2 tags transmit.
- 40000 is the bit rate for the transmission specified for Class 1 Gen 2 tags (40 Kbps).

Code division multiple access (CDMA)

The CDMA/AIC code as explained above, was run from 50 to 1000 times for 5 independent sample sets to test for repeatability. Table 3 shows the samples in this process. This test was made using 60 tags and a processing gain of 64.

Table 3. Repeatability tests with 60 tags, Gp=64 and 256 bits

Variable	Time Samples (sec)				
	Set 1	**Set 2**	**Set 3**	**Set 4**	**Set 5**
60 tags, 64 Gp, 50 sims	4.309	4.669	4.374	5.226	3.359
60 tags, 64 Gp, 100 sims	4.063	3.506	6.0293	3.875	4.809
60 tags, 64 Gp, 150 sims	4.904	4.200	5.221	5.663	4.385
60 tags, 64 Gp, 200 sims	6.119	4.190	4.956	3.793	4.968
60 tags, 64 Gp, 250 sims	4.463	3.926	4.758	4.555	5.515
60 tags, 64 Gp, 300 sims	3.987	5.336	5.617	4.339	4.912
60 tags, 64 Gp, 350 sims	5.430	4.234	5.489	5.315	5.435
60 tags, 64 Gp, 400 sims	4.966	4.692	3.920	4.872	4.729
60 tags, 64 Gp, 500 sims	4.180	4.749	4.212	4.728	4.583
60 tags, 64 Gp, 800 sims	4.500	4.789	5.021	5.211	5.454
60 tags, 64 Gp, 1000 sims	**4.437**	**4.530**	**4.442**	**4.636**	**4.696**

Dynamic framed slotted ALOHA (DFSA)

The DFSA was sampled similarly to CDMA/AIC; it also was run from 50 to 1000 times five times to get the repeatability samples. Table 4 shows the samples in this process, and for the case of the DFSA program, the repeatability starts at 800 runs. Since we need the same parameters for DFSA and CDMA, the number of runs will be 1000 for both cases.

Table 4. Repeatability tests with 60 tags, slots=64 and 256 bits

Variable	Time Samples (sec)				
	Set 1	**Set 2**	**Set 3**	**Set 4**	**Set 5**
60 tags, 64 Gp, 50 sims	0.163	0.138	0.1445	0.157	0.201
60 tags, 64 Gp, 100 sims	0.291	0.310	0.308	0.308	0.297
60 tags, 64 Gp, 150 sims	0.439	0.439	0.459	0.422	0.436
60 tags, 64 Gp, 200 sims	0.563	0.627	0.595	0.606	0.586
60 tags, 64 Gp, 250 sims	0.753	0.767	0.765	0.739	0.763
60 tags, 64 Gp, 300 sims	0.929	1.026	0.863	0.964	0.944
60 tags, 64 Gp, 350 sims	1.0425	1.045	1.043	1.040	1.057
60 tags, 64 Gp, 400 sims	1.178	1.231	1.170	1.251	1.224
60 tags, 64 Gp, 500 sims	1.534	1.581	1.509	1.492	1.559
60 tags, 64 Gp, 800 sims	**2.430**	**2.413**	**2.399**	**2.424**	**2.450**
60 tags, 64 Gp, 1000 sims	2.965	2.997	2.984	3.070	3.093

B. Time tests

For the time tests the CDMA and the DFSA programs were exposed to different parameters like noise levels, number of tags and Gp for the CDMA code and number of slots for the DFSA code. As described above, the simulation results were used to calculate the amount of transmission time required for all the tags to be successfully demodulated. This transmission time is related to three key performance parameters for the system, (1) the total time that elapses between when a reader begins interrogation of the tags and when the last of the tags is successfully demodulated, (2) the amount of energy the reader must expend to interrogate and successfully demodulate all tags, and (3) system capacity. The first parameter can be important for applications where the reader is moved to independently interrogate separate sets of tags (for example, to independently interrogate a number of pallets of merchandise spread out in a warehouse). The second parameter is important for mobile applications where a battery-powered reader will have a limited amount of energy (for example, handheld mobile readers or

small readers attached to drones operating in a warehouse). The third parameter, capacity, is related to the first two parameters, with the relationship being application-dependent

The first test involved 1000 runs of the program with a noise level of 1.577 mV^2 which is equivalent to 12 dB signal to noise ratio (SNR). The result of the test is shown in Table 5.

Table 5. First test with the CDMA and DFSA code, 1000 runs and 12 dB SNR

1000 runs, Noise = 1.577 mV^2 (12 dB SNR)								
	Gp/Slots = 32		**Gp/Slots = 64**		**Gp/Slots = 128**		**Gp/Slots = 256**	
Tags	**CDMA**	**DFSA**	**CDMA**	**DFSA**	**CDMA**	**DFSA**	**CDMA**	**DFSA**
40	2.976	1.594	2.715	1.670	2.871	2.355	3.830	3.610
60	4.738	2.330	4.552	2.411	4.614	2.900	5.097	4.120
80	6.011	3.284	5.908	3.338	4.933	3.392	6.016	4.750
100	6.667	4.008	8.239	3.897	6.246	3.932	6.255	5.316
150	10.191	6.589	10.733	6.372	12.968	6.478	16.245	6.587
200	12.023	8.217	11.362	7.995	11.307	7.823	10.915	7.780

The second test involved 1000 runs of the program with a noise level of 3.154 mV^2 which is equivalent to 9 dB signal to noise ratio (SNR). The result of the test is shown in Table 6.

Table 6. Second test with the CDMA and DFSA code, 1000 runs and 9 dB SNR

1000 runs, Noise = 1. 3.154 mV^2 (9 dB SNR)								
	Gp/Slots = 32		**Gp/Slots = 64**		**Gp/Slots = 128**		**Gp/Slots = 256**	
Tags	**CDMA**	**DFSA**	**CDMA**	**DFSA**	**CDMA**	**DFSA**	**CDMA**	**DFSA**
40	3.035	3.595	2.813	3.849	2.813	5.000	3.901	8.114
60	4.991	5.613	3.974	5.662	4.307	6.165	5.163	9.726
80	5.766	6.760	6.510	7.154	6.175	7.390	5.764	10.226
100	7.995	10.074	6.401	8.789	9.742	9.662	6.151	10.451
150	10.440	14.134	9.689	13.008	8.946	12.782	8.684	13.511
200	12.223	17.566	13.329	17.927	12.109	17.791	15.357	20.138

The third test involved 1000 runs of the program with a noise level of 6.280 mV^2 which is equivalent to 6 dB signal to noise ratio (SNR). The result of the test is shown in Table 7.

Table 7. Third test with the CDMA and DFSA code, 1000 runs and 6 dB SNR

1000 runs, Noise = 6.280 mV^2 (6 dB SNR)								
	Gp/Slots = 32		**Gp/Slots = 64**		**Gp/Slots = 128**		**Gp/Slots = 256**	
Tags	**CDMA**	**DFSA**	**CDMA**	**DFSA**	**CDMA**	**DFSA**	**CDMA**	**DFSA**
40	3.111	9296.57	2.775	2384.82	5.313	1377.57	5.499	967.56
60	4.669	3166.32	4.840	112797.80	4.324	5235.93	6.887	5222.43
80	6.180	1114.93	5.742	2347.23	5.666	2005.84	7.064	2541.99
100	7.363	3210.28	6.926	1505.69	6.051	3398.51	6.849	4116.07
150	11.769	29054.70	11.796	2565.41	9.504	5288.72	14.760	2799.07
200	12.025	5878.65	12.251	2391.71	14.568	34828.51	16.997	19724.81

The fourth test involved 1000 runs of the CDMA program only because, as shown in Table 7 (and as will be discussed in the section below), for SNR values of 6dB or lower, the time values for DFSA are too large to be practical. The noise levels were 12.52 mV^2 which is equivalent to 3 dB signal to noise ratio (SNR), and 25 mV^2 which is equivalent to 0 dB signal to noise ratio (SNR). The result of the test is shown in Table 8.

Table 8. Fourth test with the CDMA code, 1000 runs, 3dB and 0 dB SNR

CDMA - 1000 runs, Gp = 128		
Noise	**12.52 mV^2**	**25 mV^2**
Tags	**3 dB SNR**	**0 dB SNR**
40	2.816	8.119
60	4.454	9.992
80	6.688	11.179
100	6.881	11.501
150	8.706	12.303
200	11.345	14.983

VI. ANALYSIS

This section analyzes the behavior of the data depending on the number of tags, noise, processing gain (for the CDMA code), and the number of slots (for the DFSA code) under the same number of simulations. The data come from Tables 5, 6, 7 and 8. The initial value of Gp influences the efficiency of both CDMA and DFSA, and for the range of the number of tags we are considering (40 – 200) Gp = 64 and Gp = 128 are the best choices.

A. First test

From Table 5, the values of time calculated for CDMA and DFSA are practical for all cases (40, 60, 80, 100, 150, and 200 tags), with DFSA having a better time performance than CDMA. Figure 5 shows a typical plot of time values for the two systems vs. number of tags. The similarity in time values is because of the low noise that both systems work with. The high SNR (with a value of 12 dB) lets the systems demodulate the tags quickly and easily. Note that the dynamic Gp feature in the CDMA system enables it to process a total number of tags more than ten times greater than the number of tags in the fixed Gp system of Reference [8], where the practical number of tags was limited to 12.

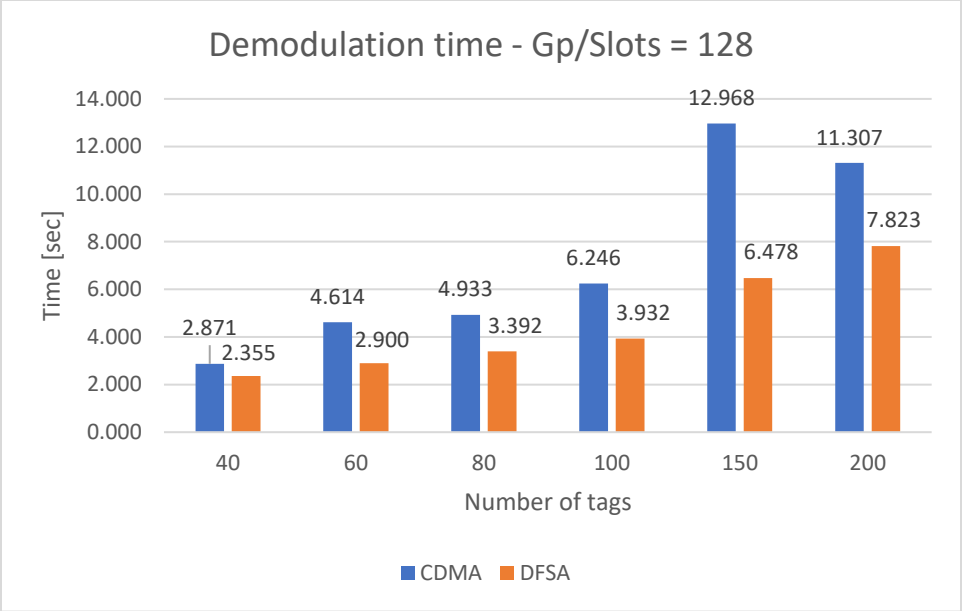


Figure 5. Time plot for CDMA/AIC (blue) and DFSA (orange) when exposed to the same conditions: 12 dB SNR, 128 Gp/slots, 1000 runs.

B. Second test

For the second test, the system's noise was increased; now, the SNR is 9 dB, and the results are shown in Table 6. When performing the codes with Gp/slots of 32 and 64 and a small number of tags, the timing is similar to 12 dB SNR in the DFSA and the CDMA case, but if we increase the number of tags, it is clear that the CDMA system starts outperforming the DFSA system. The timing for Gp/slots of 128 and 256 is different for both a small and a considerable number of tags. More than 3 seconds is the difference in time performance between CDMA and DFSA. The reduced time that CDMA has compared to DFSA shows its efficiency. Since the time of successful demodulation of the CDMA system is smaller than DFSA, more energy is saved, so CDMA is shown to be energy efficient. Furthermore, CDMA has shown to be faster in spite of all the processing and error correction in its structure. Figure 6 shows a typical

plot of time values for the CDMA and DFSA systems vs. number of tags

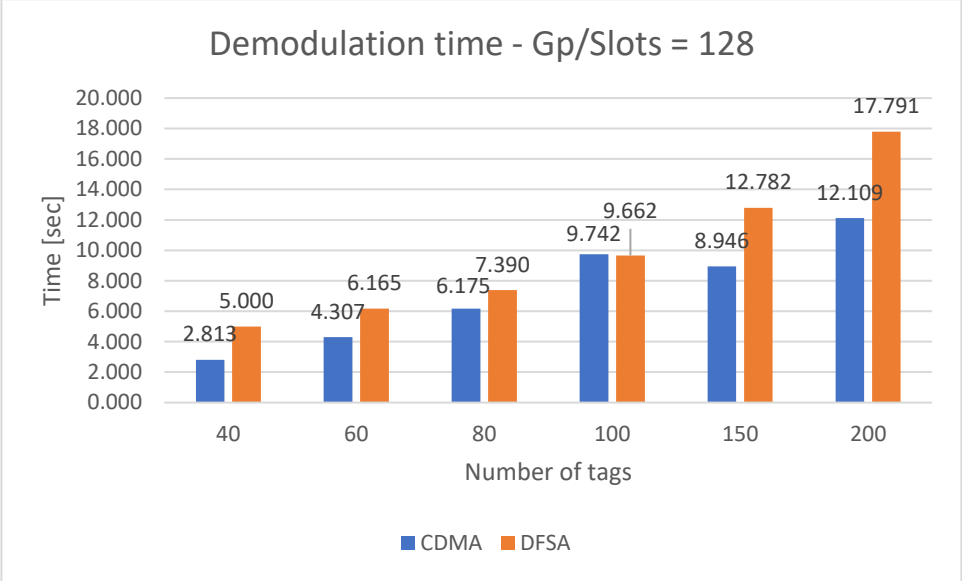


Figure 6. Time plot for CDMA/AIC (blue) and DFSA (orange) when exposed to the same conditions: 9 dB SNR, 128 Gp/slots, 1000 runs.

C. Third test

In the third test, the SNR decreases to 6 dB, which means that noise of 6.280 mV^2 is used in both systems. Table 7 shows that as more noise is introduced into the system DFSA is not able to operate successfully anymore. For almost all the cases with a small number of tags, the time is greater than 1000 seconds. Comparing these values with CDMA, we can see they are very different; while CDMA takes a couple of seconds to demodulate all the tags successfully, the DFSA system spends thousands of seconds in the same process. This difference in time happens because of CDMA's error correction features; spreading the signal over the channel helps to resolve collisions and to correct errors. These two characteristics make the CDMA process fast and efficient, even in

noisy scenarios. In the case of DFSA, retransmission, when there is an unsuccessfully demodulated tag, is often not helpful because the retransmission will often also have an error. If that error is retransmitted once and again, the system will increase the slot size, and time and efficiency will decrease. In noisy scenarios increasing the number of slots for a DFSA case does not resolve the error in the retransmission.

Figure 7 shows the robust behavior of CDMA/AIC when operates for all tested number of tags.

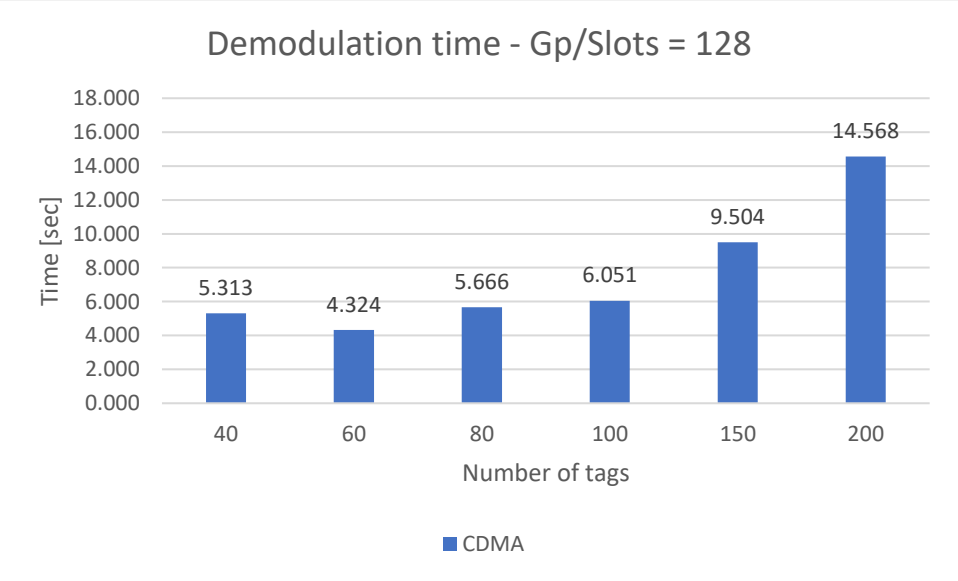


Figure 7. Time plot for CDMA/AIC with dynamic processing gain when exposed to 6 dB SNR, initial Gp = 128, 1000 runs.

D. Fourth test

The unstable behavior of DFSA for SNR of 6 dB shows that it is not a noise-proof system, but with CDMA, even if the noise keeps increasing to 12.52 mV^2 (3 dB SNR)

and 25 mV^2 (0 dB SNR), the successful demodulation of the tags is still possible, as shown in Table 8. This is an important feature for applications where the noise levels are high and also for mobile systems that may need to operate with low transmitted power.

The CDMA system was tested under noise conditions of 3 dB and 0 dB SNR. In spite that the noise levels are very high, the system remains stable and can still successfully demodulate all the tags in a small amount of time. A great amount of energy efficiency can be achieved with this CDMA/AIC with dynamic Gp approach.

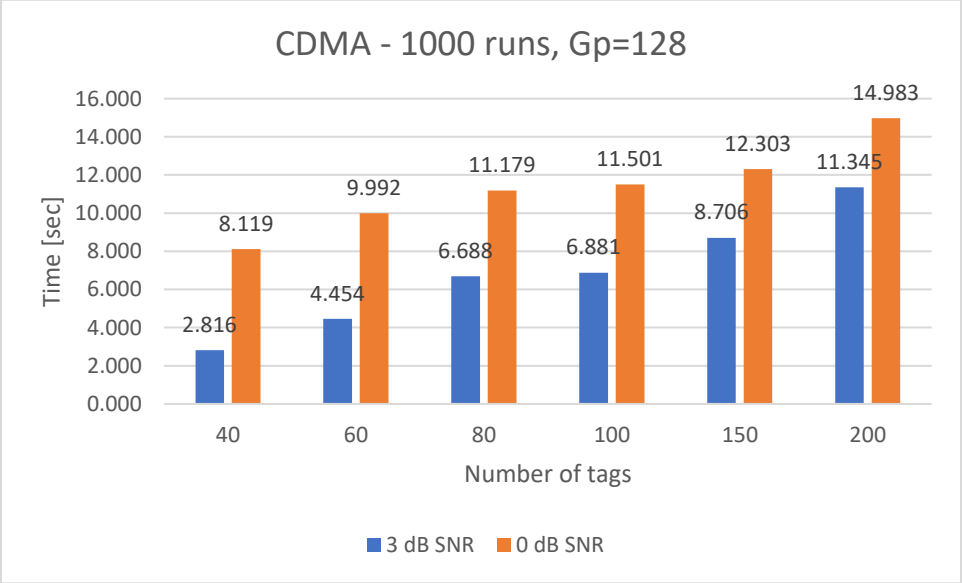


Figure 8. Time plot for CDMA/AIC when exposed to: 3 dB SNR and 0 dB SNR, initial Gp=128, 1000 runs.

Table 9 shows the values of the time for the CDMA/AIC system with dynamic Gp at different noise levels and the same initial Gp, and Figure 9 compares the 3 cases of noise levels shown in Table 9. The time values for the 3 cases are very close to each other and this shows how stable the system is when exposed to different noise levels.

Table 9. Successful demodulation times for a CDMA/AIC system with dynamic processing gain, for different noise levels and the same initial Gp=128

1000 runs, CDMA/AIC (only), Gp = 128			
Tags	**1.577 V² 12 dB SNR**	**3.156 V² 9 dB SNR**	**6.280 V² 6 dB SNR**
40	2.871	2.813	5.313
60	4.614	4.307	4.324
80	4.933	6.175	5.666
100	6.246	9.742	6.051
150	12.968	8.946	9.504
200	11.307	12.109	14.568

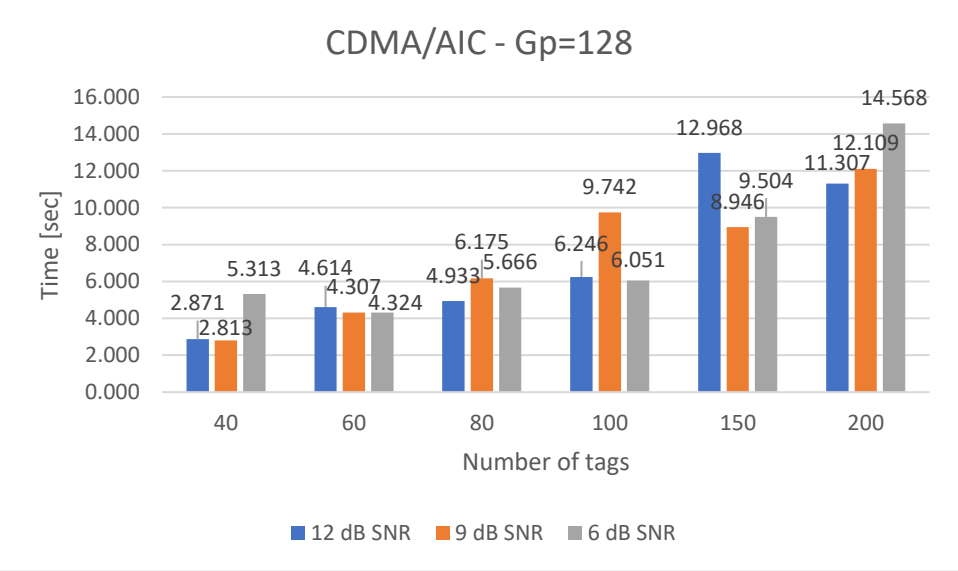


Figure 9. Time performance of CDMA/AIC when exposed to different noise levels (12 dB, 9 dB, 6 dB SNR) and a Gp = 128

VII. CONCLUSIONS

Many factors make RFID systems vulnerable to collision problems, especially when they have passive tags. Shadowing, noise, and the near-far problem continuously affect the wireless transmission of this type of system, making them unreliable and highly energy-consuming. To deal with the collision problem, much research has been developed in anti-collision algorithms using TDMA and CDMA modulation techniques; some with fixed and others with dynamic parameters, but all the time with the same objectives: successful transmission, and energy efficiency.

The collision problem in RFID systems motivated this work, which implemented an anti-collision algorithm with a CDMA/AIC technique and dynamic processing gain. The dynamization of the Gp is a new topic, and it is considered an important parameter that helps the CDMA RFID system to save energy during the demodulation process.

For the creation of the anti-collision algorithm, MATLAB was used, a powerful software tool that lets the researchers make high mathematical calculations required in communication systems. For the research, two codes, CDMA/AIC and DFSA, were run under the same parameters of noise, the number of tags, and Gp/slots. Then, the outputs from the programs were compared to understand their behavior under specific conditions.

The DFSA, modified from a previous reference [8] in order to model dynamic behavior, shows a very efficient performance when exposed to a low-noise environment. It demonstrated that the time it takes to demodulate all the system tags successfully is better than the CDMA/AIC exposed to the same low noise condition. However, when the noise was increased (SNR of 6 dB or less), the system's instability was shown, and this technique could not correct the errors anymore. With every retransmission, many of the

errors were retransmitted, the size of the frame became bigger (without solving the problem), and significant undesired delays were introduced in the system.

On the other hand, the CDMA code with dynamic processing gain performed better when exposed to noisy environments (or environments with low transmitted power) and with many tags. Although its performance in high SNR environments was not better than the DFSA, the timing was not larger than a couple of seconds. When the CDMA was exposed to applications and environments that produced medium or low SNR, its time was reasonable and a thousand seconds less than the DFSA. The stability of CDMA is achieved because of its error correction feature. Using spread spectrum enables both error correction and collision control. It will correct the error as soon as it appears in the system, saving retransmission energy and time in the next run. Unlike DFSA, CDMA/AIC with Dynamic Gp can successfully run-in high noise environments such as factory floors, and, for low noise environments, it can allow the transmitted power from the reader to be reduced, greatly improving power efficiency and, thereby, increasing operating time between charges for mobile readers. For more alternative applications, such as mounting readers on drones in warehouses, improvements in power efficiency can also translate into reduced weight for the readers.

It is essential to highlight that the dynamic processing gain is the third parameter that makes this system strong (besides CDMA and AIC); the size of Gp depends on the number of successfully demodulated tags from the previous run. Automatically resizing the Gp ensures that the system will not use excessive time or energy in its next run. This feature makes the CDMA/AIC with dynamic processing gain an energy and time-saving system.

After the tests made for this thesis and with all the results obtained from the simulations and calculations, it was proven that CDMA is a robust system when exposed to noisy environments. Its characteristics can be enhanced if an AIC and dynamic Gp algorithms are added to its structure. Comparing reference [8], which was the starting point for this work, this thesis made the fixed parameters flexible. Gp was dynamic depending on the number of successfully demodulated tags in the previous run, and the number of tags was increased from 9 - 12 in [8] to 40 – 200 (and can still grow more), and the noise levels were changed for the tests.

VIII. FUTURE RESEARCH SUGESTIONS

From the results of this thesis and showing that CDMA/ AIC with dynamic processing Gain is a robust system, there is some future research that can be implemented.

With this thesis's results, accuracy tests can be run comparing the CDMA and DFSA systems. Of course, their time performance is already known, but it would be advantageous to see the percentage of tags successfully demodulated within a fixed amount of time. This information could be useful for applications where speed is essential, and a very high level of accuracy may not be needed.

The CDMA/AIC code with dynamic processing gain is a very robust system, so its processing time is very high, especially with a large number of tags. Therefore, it would be beneficial if the code is optimized in a way that successfully demodulates more than 300 tags in a couple of minutes. For this case, when a low processing gain (32) and a large number of tags (200) are introduced, it takes around 45 minutes of processing time in the simulation to demodulate all the tags successfully. In a similar way, it may be possible to create an algorithm allowing the system to quickly evaluate its environment and estimate the optimal initial value for processing gain.

A level of conventional linear error correction coding can be introduced for the DFSA system, which may provide that system with more durability in low-SNR environments and may be worth the overhead introduced by the coding. Note, however, that the error-correcting power of the code will be fixed, while the error-correcting power of CDMA/AIC with dynamic Gp is efficiently adjusted by the changes in Gp.

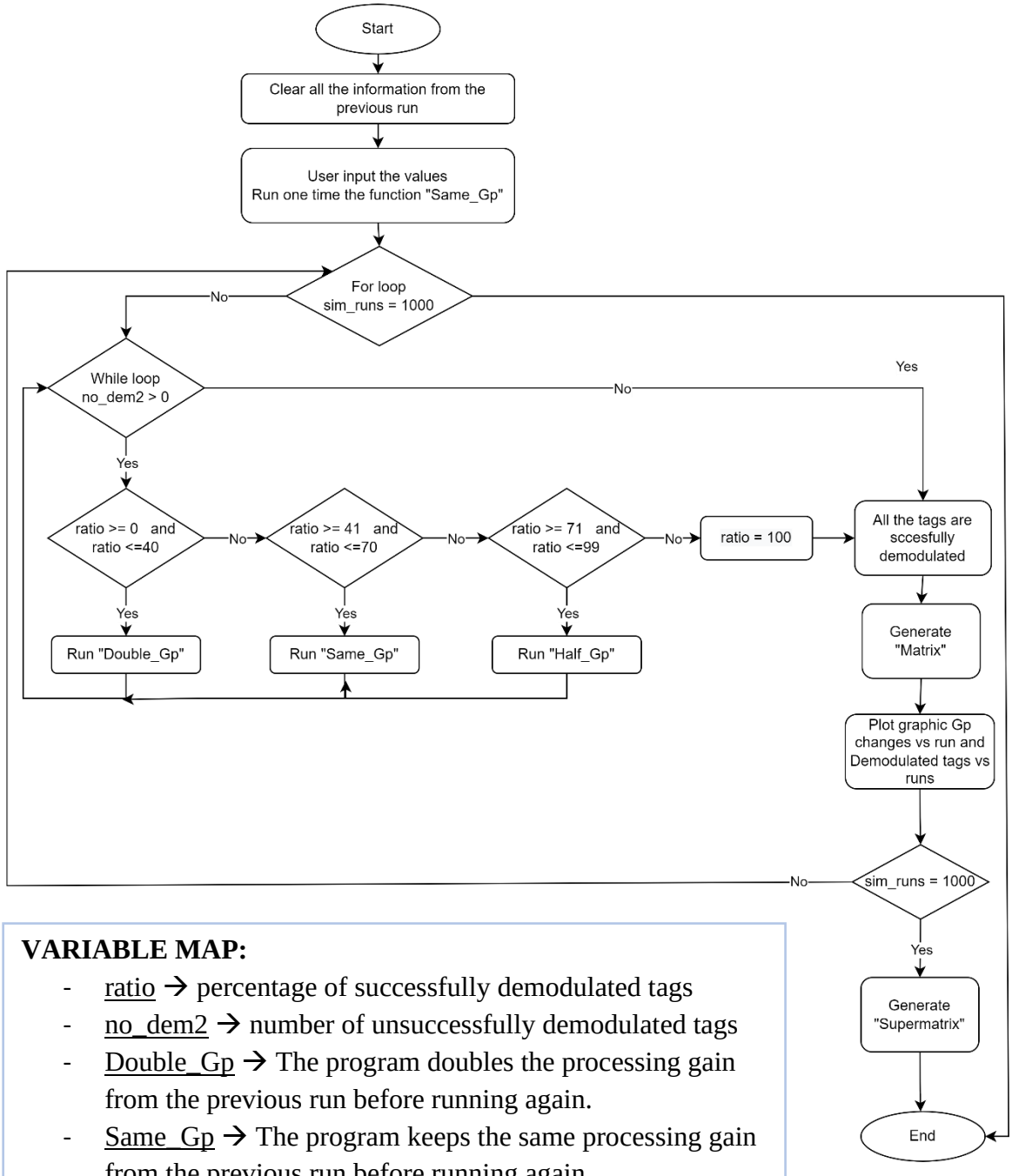
Additional research is necessary to optimize the thresholds for halving and

doubling Gp. The thresholds will be dependent on SNR and the number of tags in the system.

The developed system is immune to noise, and an application that keeps the tools and materials in workshops can be proposed. Passive tags are put on the tools, and the readers will detect all the tools inside plastic toolboxes to ensure that all the tool sets are complete. The system can read all the tags inside the box at the same time and give a response of complete tools or missing tools.

APPENDIX SECTION

Appendix A: Flow diagram



VARIABLE MAP:

- ratio → percentage of successfully demodulated tags
- no_dem2 → number of unsuccessfully demodulated tags
- Double_Gp → The program doubles the processing gain from the previous run before running again.
- Same_Gp → The program keeps the same processing gain from the previous run before running again.
- Half_Gp → The program reduces the processing gain to the half from the previous run before running again.

Appendix B: CDMA/AIC main code

```
%-------------------------------------------------------------------------
% MAIN CDMA PROGRAM
%-------------------------------------------------------------------------
%Before starting all the registers will be cleaned up.
clear
clc

%-------------------------------------------------------------------------
% INPUT OF THE MAIN VARIABLES OF THE PROGRAM
%-------------------------------------------------------------------------
no_oftags=input('Input the number of tags:   ');
no_ofbits=input('Input the number of bits:   ');
sim_runs=input('How many times the program will run?   ');
processing_gain=input('Input the processing gain:    ');
pointr=0; % Pointer for building the super matrix
fprintf('\n');
fprintf('------------------------------------------------------------------');
fprintf('\n');

%-------------------------------------------------------------------------
% RESET OF THE VARIABLES
% The following variables keep the value of the original input values of
% the variables to work with them in the different runs
%-------------------------------------------------------------------------
reset_tags = no_oftags;
reset_gp = processing_gain;

%-------------------------------------------------------------------------
%MULTIPLE SIMULATIONS LOOP
%-------------------------------------------------------------------------
for n=1:sim_runs
    simcount = n;
    no_oftags = reset_tags;
    no_dem2=reset_tags;
    processing_gain = reset_gp;
    sims = sim_runs;

%-------------------------------------------------------------------------
%INITIALIZATION OF THE VARIABLES FOR THE OUTPUT MATRIX
%-------------------------------------------------------------------------
iterationUntilCompletion=1; %INITIALIZATION OF THE COUNTER FOR THE %ITERATIONS
iter_change=zeros(1,1000); % MATRIX OF ZEROS FOR STORING THE VALUES OF THE
%ITERATION WHILE THEY CHANGE

sim=0;
sim_change=zeros(1,1000); % MATRIX OF ZEROS FOR STORING THE VALUES OF THE
%NUMBER OF SIMULATIONS WHILE THEY CHANGE

i=0;
Gp_change=zeros(1,1000); % MATRIX OF ZEROS FOR STORING THE VALUES OF THE GP
%WHILE THEY CHANGE
```

```
fail=0;
fail_change=zeros(1,1000); % MATRIX OF ZEROS FOR STORING THE VALUES OF THE
%UNSUCCESSFUL DEMODULATED TAGS WHILE THEY CHANGE

success=0;
success_change=zeros(1,1000); % MATRIX OF ZEROS FOR STORING THE VALUES OF %THE
SUCCESSFUL DEMODULATED TAGS WHILE THEY CHANGE

% The elapsed time starts after the first test
tic;

%--------------------------------------------------------------------------
% START OF THE PROGRAM
%--------------------------------------------------------------------------
%This is the first round of the program with the original program, it will
%run 1 time with desired
% Gp and will wait for 3 seconds before going to the while loop.
Same_Gp
out = ['Percent of succesful demodulated messages:   ', num2str(ratiostr),
'%'];
disp (out)
fprintf('\n');
dem_tags = ['The number of tags succesfully demodulated is:   ',
num2str(calc)];
disp (dem_tags)
fprintf('\n');
nodem_tags = ['The number of tags unsuccesfully demodulated is:   ',
num2str(no_dem)];
disp (nodem_tags)
fprintf('\n');
Gp=processing_gain;
fprintf('Initial the Gp.')
fprintf('\n');
processing_gain;
fprintf('\n');
fprintf('--------------------------------------------------------------------');
fprintf('\n');
pause(3)
sims=sims;

% no_dem2 is the variable that indicates the number of unsuccessful
% demodulated tags, and meanwhile that is greater than 0, the loop
% continues running the code.
while (no_dem2 > 0)

% The variable fout is the percentage of demodulated tags per run, so here %
he have 3 cases: the double of the Gp from the previous run, the same Gp % as
the previous run, and the half of the Gp from the previous run.
% If the percentage of successful demodulated tags is between 0 and 40%, %the
Gp is going to double in the next run.
% If the percentage of successful demodulated tags is between 40% and 70%,
%the Gp is going to remain the same in the next run.
% If the percentage of successful demodulated tags is between 70 and 99%, %the
Gp is going to be the half in the next run.
```

```
% If the percentage of successful demodulated tags is 100%, the loop breaks
%and the program ends.

    iterationUntilCompletion= iterationUntilCompletion + 1 ; %COUNTER FOR %THE
ITERATIONS
    iter_change(iterationUntilCompletion)=iterationUntilCompletion;

    sim=sim+1; %COUNTER FOR THE NUMBER OF SIMULATIONS
    sim_change(sim)=sims;

    i=i+1; %COUNTER FOR CHANGES OF THE GP
    Gp_change(i)=Gp;

    fail=fail+1; %COUNTER FOR THE UNSUCCESSFUL DEMODULATED TAGS
    fail_change(fail)=no_dem2;

    success=success+1; %COUNTER FOR THE SUCCESSFUL DEMODULATED TAGS
    success_change(success)=ncalc;

    ratio = 100*ncalc/(ncalc+no_dem2);
    ratiostr = ratio;
%--------------------------------------------------------------------------
%CONDITION FOR DOUBLING THE PROCESSING GAIN
%--------------------------------------------------------------------------
    if ratio >= 0 && ratio <= 40
        Double_Gp % Double Gp
        out = ['Percent of successful demodulated messages:   ',
num2str(ratiostr), '%'];
        disp (out)
        fprintf('\n');
        dem_tags = ['The number of tags successfully demodulated is:   ',
num2str(calc)];
        disp (dem_tags)
        fprintf('\n');
        nodem_tags = ['The number of tags unsuccessfully demodulated is:   ',
num2str(no_dem)];
        disp (nodem_tags)
        fprintf('\n');
        fprintf('Doubling the Gp.')
        fprintf('\n');
        processing_gain;
        Gp=processing_gain; %Take the value of Gp for the matrix
        fprintf('\n');
        fprintf('-------------------------------------------------------------
----');
        fprintf('\n');
        pause (3)
%--------------------------------------------------------------------------
%CONDITION FOR KEEPING THE SAME PROCESSING GAIN
%--------------------------------------------------------------------------
    elseif ratio > 40 && ratio <= 70
        Same_Gp % Same Gp
        out = ['Percent of successful demodulated messages:   ',
num2str(ratiostr), '%'];
        disp (out)
```

```
        fprintf('\n');
        dem_tags = ['The number of tags successfully demodulated is:   ',
num2str(calc)];
        disp (dem_tags)
        fprintf('\n');
        nodem_tags = ['The number of tags unsuccessfully demodulated is:   ',
num2str(no_dem)];
        disp (nodem_tags)
        fprintf('\n');
        fprintf('Keeping the same Gp')
        fprintf('\n');
        processing_gain;
        Gp=processing_gain; %Take the value of Gp for the matrix
        fprintf('\n');
        fprintf('-------------------------------------------------------------
----');
        fprintf('\n');
        pause (3)
%-------------------------------------------------------------------------
%CONDITION FOR HALVING THE PROCESSING GAIN
%-------------------------------------------------------------------------
    elseif ratio > 70 && ratio <= 99
        Half_Gp % Half Gp
        out = ['Percent of successful demodulated messages:   ',
num2str(ratiostr), '%'];
        disp (out)
        fprintf('\n');
        dem_tags = ['The number of tags successfully demodulated is:   ',
num2str(calc)];
        disp (dem_tags)
        fprintf('\n');
        nodem_tags = ['The number of tags unsuccessfully demodulated is:   ',
num2str(no_dem)];
        disp (nodem_tags)
        fprintf('\n');
        fprintf('Reducing the Gp to the half.')
        fprintf('\n');
        processing_gain;
        Gp=processing_gain; %Take the value of Gp for the matrix
        fprintf('\n');
        fprintf('-------------------------------------------------------------
----');
        fprintf('\n');
        pause (3)
%-------------------------------------------------------------------------
%CONDITION FOR 100% SUCCESFULLY DEMODULATED TAGS
%-------------------------------------------------------------------------
    else
        fprintf('-------------------------------------------------------------
----');
        fprintf('\n');
        fprintf('End of the Program')
        fprintf('\n');
    end
 end
```

```
toc;

aa=iterationUntilCompletion; % KEEP THE VALUE AS A NUMBER BEFORE MAKING IT A
STRING FOR DISPLAYING IT
fprintf('\n');
iteration_num = ['Number of times the Gp changes: ', num2str(aa)];
disp (iteration_num)
fprintf('\n');

%--------------------------------------------------------------------------
% THE OUTPUT MATRIX SHOWS:
%   - FIRST COLUMN: SIMULATION NUMBER
%   - SECOND COLUMN: PARTICULAR ROUND
%   - THIRD COLUMN: PROCESSING GAIN
%   - FOURTH COLUMN: NUMBER OF UNSUCCESSFUL DEMODULATED TAGS
%   - FIFTH COLUMN: NUMBER OF SUCCESSFUL DEMODULATED TAGS
%--------------------------------------------------------------------------

matrix=zeros(1000,5); %MATRIX FOR THE OUTPUT VALUES:RUNS, ROUNDS, GP, SUCCESS
AND FAIL

iter_change(iterationUntilCompletion)=aa;
iter_change(1,1)=1;
iter_change = transpose(iter_change);%SHOWS THE CHANGES OF THE SIMULATION
NUMBER AS A VERTICAL VECTOR

sim_change(sim+1)=sim_runs;
sim_change(1,1)=1;
sim_change = transpose(sim_change);

Gp_change(i+1)=Gp;
Gp_plot= Gp_change; %Vector of Gp to be plotted
Gp_plot( :, ~any(Gp_plot,1) ) = [];
Gp_change = transpose(Gp_change); % SHOWS THE CHANGES OF THE Gp AS A VERTICAL
VECTOR

fail_change(fail+1)=no_dem2;
fail_change=transpose(fail_change); % SHOWS THE CHANGES OF THE UNSUCCESSFUL
TAGS AS A VERTICAL VECTOR

success_change(success+1)=ncalc;
success_plot= success_change; %Vector of demodulated tags to be plotted
success_plot( :, ~any(success_plot,1) ) = [];
success_change=transpose(success_change); % SHOWS THE CHANGES OF THE
UNSUCCESSFUL TAGS AS A VERTICAL VECTOR

fprintf('\n');
fprintf('----------------------------------------------------------------');


matrix(:,1)=iter_change; %STORING THE VALUES OF THE SIMULATION NUMBER IN %THE
FIRST ROW OF THE MATRIX
matrix(:,2)=sim_change; %STORING THE VALUES OF THE ACTUAL RUN IN THE SECOND
%ROW OF THE MATRIX
```

```
matrix(:,3)=Gp_change; %STORING THE VALUES OF THE Gp IN THE THIRD ROW OF %THE
MATRIX
matrix(:,4)=fail_change; %STORING THE VALUES OF THE UNSUCCESSFUL %DEMODULATED
TAGS IN THE FOURTH ROW OF THE MATRIX
matrix(:,5)=success_change; %STORING THE VALUES OF THE SUCCESSFUL %DEMODULATED
TAGS IN THE FIFTH ROW OF THE MATRIX

%--------------------------------------------------------------------------
% WORKING ON THE SECOND COLUMN OF THE MATRIX
% Overwriting method
%--------------------------------------------------------------------------
for jx=1:aa
    matrix(jx,2) = simcount;
end

%--------------------------------------------------------------------------
%OUTPUT MATRIX
%--------------------------------------------------------------------------
matrix; % SHOWS THE OUTPUT MATRIX
matrix( ~any(matrix,2), : ) = []  %Get rid of the zero rows and shows the
final matrix
fprintf('\n');
fprintf('-----------------------------------------------------------------');
fprintf('\n');

success_plot = transpose(matrix(:,5));

%--------------------------------------------------------------------------
%OUTPUT PLOTS
%--------------------------------------------------------------------------
subplot(2,1,1);
plot(Gp_plot,'r-o','LineWidth',1.5)
grid on
title('Change in Processing Gain')
xlabel('Number of runs')
ylabel('Processing gain')

subplot(2,1,2);
plot(success_plot,'b-o','LineWidth',1.5)
grid on
title('Successfully demodulated tags')
xlabel('Number of runs')
ylabel('Number of tags')


%--------------------------------------------------------------------------
%SUPERMATRIX GENERATION
%--------------------------------------------------------------------------
    for i=1:aa
        for j=1:5
            supermatrix(i+pointr,j) = matrix(i,j);
        end
    end

    pointr = pointr + aa;
```

```
end %end of the loop for each simulation

pause(5)
supermatrix

% This line of code export the data from Matlab to excel the argument of %the
function is:
%   - supermatrix.xlsx: the name of the excel file that has to be in the %same
folder as the programs
%   - supermatrix: the matrix to be printed in the excel file
%   - Sheet 1: sheet in which the data will be exported
%   - A2: cell since where the data will be printed
xlswrite('supermatrix.xlsx', supermatrix, 'Sheet 1', 'A2')
```

Appendix C: CDMA/AIC Function 1 (Double processing gain)

```
%-------------------------------------------------------------------------
% FUNCTION 1 CDMA PROGRAM - DOUBLE GP
% TAKEN FROM REFERENCE [8]
%-------------------------------------------------------------------------

%Parameters from previous run
no_oftags=no_dem2;
no_ofbits=256;
sim_runs=sims;
processing_gain=2*processing_gain;

%-------------------------------------------------------------------------
% NOTE THAT THE ELAPSED TIME STARTS HERE, THE TIME THAT THE USER SPEND
% INPUTING VARIABLES DOES NOT COUNT.
%-------------------------------------------------------------------------
tic; %tic-toc function to measure elapsed time

tau_fail=0; %Diagnostic variable
spread_overpowered=0; %Diagnostic variable
successful_demodulation=0; % Counter for number of tag messages
noise_power=1.577; %Scalar value of noise power (sigma squared) in (mV^2)

%The loop below(using the variable iruncount) spans most of the program
for iruncount=1:1
    tag_arr = randi(0:1, no_oftags, no_ofbits); %Generate array containing
    % original binary data for all tags
    scm = hadamard(processing_gain); %THERE IS NO NEED TO TRANSPOSE THE
    % MATRIX
    %The rows of scm are now orthogonal spreading codes
    %The next loop of 6 lines randomizes (or "scrambles") the rows of scm,
    % which helps remove correlation
    %between adjacent rows and will subsequently simplify the process of
    % having each tag randomly choose a spreading code
    for j=1:no_oftags
        scramble=randi(processing_gain); %NO_OFTAGS TIMES GET RANDOM
        % NUMBERS BETWEEN 1 AND 64

        %For no_oftags > Gp
        if j>processing_gain
            jm = mod (j,processing_gain)+1;
        else
            jm=j;
        end
        temp=scm(jm,:); %TAKES THE FIRST ROWN OF 64 BITS

        scm(j,:)=scm(scramble,:);
        scm(scramble,:)=temp; %SCM IS THE NEW MATRIX WHITH SCRAMBLED ROWS
    end

    sc_t=scm;
    sc_t(sc_t == -1)=0; % sc_t now contains the different spreading codes
```

```
% in its rows, converted to 1s and 0s instead of 1s and -1s
%The loops below create the matrix out_array. Each row of out_array
% contains the spread data corresponding to one of the tags.
%Because the rows of scm (and therefore sc_t) have been "scrambled",
% the process simulates each tag randomly selecting a spreading code.
out_array=zeros(no_oftags,processing_gain*no_ofbits);

for p=1:no_ofbits   %P GOES FROM 1 TO 256
    for n=1:no_oftags %n GOES FROM 1 TO 9
        for m =1:processing_gain %m GOES FROM 1 TO 64
            point=m+(p-1)*processing_gain;
            out_array(n,point)=xor(tag_arr(n,p),sc_t(n,m)); %PN made
        end
    end
end

%The next section produces a Rayleigh distribution for the relative
% amplitude of each tag's transmission.
%This simulates the effect of multipath and shadowing
tau=zeros(1,no_oftags);
sigma=1;
for n=1:no_oftags
    tau(1,n)= sigma*sqrt(-2*log(1-rand(1))); %Numerical explanation in
    % separate page
end


%The next section creates the analog transmitted signal for each tag
%Nominally, 5 millivolts is used to represent a "1" and -5 millivolts
% is used to represent a "0", but each tag's signal must then be
%multiplied by the "tau" to include the effects of multipath and
% fading. Each row of the matrix volts_forall will contain the analog
%signal corresponding to one tag.
%This signals are PAM with rectangular pulses. We may need to modify
% them. Avg signal power = (5 millivolts)^2
volts_forall=zeros(no_oftags, no_ofbits*processing_gain);
for n = 1:no_oftags
    volts_forall(n,:)=(-5 + 10*out_array(n,:))*tau(1,n);
end
%Since all the signals are transmitted simultanously, the total
% transmitted signal is the sum of all the individual signals. The
% section below creates analog_signal, which is the aggregate
% transmitted signal

volts_add=zeros(1,point);
for n = 1:no_oftags
    volts_add(1,:)=volts_add + (volts_forall(n,:));
end
transmitted_signal=volts_add; %this is the total signal seat over the
% channel

%Now add the noise to the analog transmitted signal. Channel
% attenuation could be added, too, but won't change the analysis as
%long as received SNR is the parameter used to evaluate performance
noise_db = 10*log10(noise_power); %remember that signal and noise
```

```
    % power measurements are given in millivolts^2
    x_noise=wgn(point,1,noise_db); %WHITE GAUSSIAN NOISE FUNCTION -
    % VERTICAL VECTOR - LENGTH=16384, 1 ROW
    x_noise=transpose(x_noise);
    received_signal=transmitted_signal + x_noise; %Received signal has not
    % been attenuated (gamma=1) but noise had been added.

    %Now start demodulation,either of the received signal when (z=1) or of
    % the received signal after an AIC loop (when z>1)
    New_aggregatedSignal=received_signal; %Before AIC is applied
    % New_aggregatedSignal will be the same as the received signal
    %but not after AIC is applied.

    %The large loop below (using the variable z) demodulates, despreads,
    % finds the tag with the strongest signal, extracts the data
    %from the tag and then uses AIC to remove the effect of the strongest
    % tag
    for z=1:no_oftags
        %First step: demodulate the received signal. Later we may want to
        % use different variable to represent the analog received signal
        % and the demodulated received signal
        voltsadd_val(1:point)=(New_aggregatedSignal(1:point) +
abs(New_aggregatedSignal(1:point)))/2;
        voltsadd_val(voltsadd_val>0) = 1; %voltsadd_val is now the
        % demodulated, spread signal at the receiver (1s and 0s)

        %Second step: despreads the first bit of the received signal using
        % each possible spreading code (we are only dispreading the first
        % chip because that information will be sufficient to tell us which
        % tag sent the strongest signal)
        %The first "chip" is the first 64 bits of the of the spread signal.
        % After spreading it will correspond to one bit of information
        add_signal = voltsadd_val(1:processing_gain); %add_signal is the
        % first chip of received signal
%-------------------------------------------------------------------------
% HERE THE PARAMETER OF "ZEROS" MATRIX WAS CHANGED, WE DO NOT NEED A 64X64
% MATRIX ANYMORE, THE #TAGS WILL DETERMINE THE NUMBER OF ROWS
%-------------------------------------------------------------------------
        despreading_forall = zeros (no_oftags, processing_gain);

        for n = 1:no_oftags
            despreading_forall(n,:) = xor(add_signal, sc_t(n,:));
        end
        %Each row of despreading_forall now contains the first chip of the
        % received signal xor-ed with one of the possible spreading codes
        despread_results=despreading_forall;

        %Third step: determine which spreading code produced the chip that
        % is most consistent (i.e., has the most 1s or 0s). That the code
        %will correspond to the strongest tag.
        counting_rows = zeros(no_oftags,2);
        for n = 1:no_oftags
            counting_rows(n,1) = nnz(despread_results(n,:) == 1);
            counting_rows(n,2) = nnz(despread_results(n,:) == 0);
        end
```

```
        %Each row of counting_rows contains consistency information for a
        % particular spreading code. The first element in the row contains
        %the number of 1s, the second element contains the number of 0s
        %Now identify the spread tags with the greatest consistency.
        subtract_1 = zeros(no_oftags,1);
        for n = 1:no_oftags
            subtract_1(n,1) = abs(counting_rows(n,1)- counting_rows(n,2));
        end
        highest_num = max(subtract_1); %higest_num= maximum differential of
        % 1s and 0s. The higher this value, the greater the consistency
        %disp ('highest_num value is: ',num2str(highest_num)) %SHOW THE
        % VALUE TO CHECK IF IT IS HIGH
        guess_winner = find(subtract_1 == highest_num);
        %guess_winner is a 1-column array containing the numbers of all
        % tags producing the greatest consistency (i.e., the strongest
        %tags). The first element in this array will be used as the
        % strongest tag.
        %Note that there may be multiple tags with the same, greatest
        % consistency.

        %In most cases, selecting any one of these tags for our first pass
        % through AIC will allow us to successfully extract the tags data.
        %However, if an error occurs we want to be able to try again using
        % each of the other "greatest consistency" tags to see if we can
        %extract that tag's data without error
        sz = size(guess_winner);
        n_strong=sz(1); %n_strong is the number of tags with the greatest
        % consistency
        %Create a loop for strongest tag
        for n_aic = 1:n_strong
            guess_winner_despread_code =
despread_results(guess_winner(n_aic),:);
            %guess_winner_despread_code is the despreads code corresponding
            % to the potentially strongest tag (i.e., one of the tags with
            %the greatest consistency)
            guess_winner_spreading_code = sc_t(guess_winner(n_aic),:);
            %guess_winner_despread_code is the spreading code corresponding
            % to the potentially strongest tag

            %Fourth step: Now that a potentially strongest tag has been
            % identified, despreads all the received data using only the
            %spreading code from the potentially strongest tag.
            for p = 1:no_ofbits
                for m = 1:processing_gain
                    point = m +(p-1)*processing_gain;
                    despread_strong_tag(point) =
xor(voltsadd_val(point),guess_winner_spreading_code(m));
                end
            end

            %Fifth step: extract the original unspread data from the
            % potentially strongest tag
            for p = 1:no_ofbits
```

```
                counting_ones=0;
                counting_zeros=0;
                for m = 1:processing_gain
                    point = m+(p-1)*processing_gain;
                    if despread_strong_tag(point) == 1
                        counting_ones = counting_ones + 1;
                    else
                        counting_zeros = counting_zeros + 1;
                    end
                    if counting_ones >= counting_zeros
                        data(p) = 1;
                    else
                        data(p) = 0;
                    end
                end
            end %The array data now contains the extracted data from the
            % potentially strongest tag.

            %Sixth step: Verify that the extracted data is correct. In
            % practical applications this verification will be done using a
            %small Cyclic Redundancy Check (CRC) code. Since the CRC will
            % be necessary, whether the system uses conventional slotted
            %ALOHA or CDMA, it's easier in this simulation to just check
            % the extracted data against the original data. This shortcut
            %won't change the performance comparison of the slotted ALOHA
            % system versus CDMA.

            %If extracted data is correct, the code below will set
            % datacheck will equal 0. If the extracted data has one or more
            %errors, datacheck will be equal to 1.

%-------------------------------------------------------------------------
% THIS IS THE COMPARISON OF THE ORIGINAL AND DEMODULATED MESSAGE
%-------------------------------------------------------------------------
            %COMPARES BIT BY BIT THE ELEMENTS OF BOTH ARRAYS.
            datacheck=0;
            cc=1:no_ofbits; % CC IS A NEW VARIABLE IN THE CODE THAT
            % REPLACES THE FOR LOOP, IT TAKES VALUES FROM 1 TO 256
            y=tag_arr(guess_winner(n_aic),cc);
            x=data(cc);
            if x == y
                datacheck=0;
                %'Successful demodulation'
            else
                datacheck=1;
                %'Error in demodulation'
            end


            if datacheck == 1
                continue  %if extracted data has an error, go to the end
                % of the loop and start over with another potentially
                              %strongest tag
            end
           successful_demodulation = successful_demodulation + 1;
```

```
            %Seventh step: estimate the amplitude of the received signal
            % corresponding only to the strongest tag.

            %First, recreate the spread data corresponding to the strongest
            % tag
            for p = 1:no_ofbits
                for m = 1:processing_gain
                    point = m + (p-1)*processing_gain;
                    strong_tag(point) = xor(data(p),
guess_winner_spreading_code(m));
                end
            end

            %Second, estimate the amplitude of the received signal
            % corresponding only to the strongest tag. The first part of
            % the estimate involves determining the max and min values of
            % the received signal for those bits where strong_tag=1. We
            % start by initializing some variables.

            %The variable temphigh is initialized to -100 instead of zero,
            % because in rare cases all appropriate values of v may be
            %negative but they will not all be less than -100. templow is
            % initiated to 100 instead of 0 because in rare cases all
            %appropriate values of v may be positive but they will not be
            % grater than 100. Later we may want to refine this code.

                %If messages are long enough, it's reasonable to assume
                % that sometime during transmission the strongest tag is
                %transmitting a "1" and all other tags are also
                % transmitting a "1" (high value), and that at some other
                %time the strongest tag is transmitting a "1" but all other
                % tags are transmitting a "0" (low value). The amplitude of
                %the strongest signal is ("high"+"low")/2

            onescount=0;
            temphigh=0;
            templow=0;
            highpointer=-100;
            lowpointer=100;

            for p = 1:no_ofbits
                for m = 1:processing_gain
                    point = m + (p-1)*processing_gain;

                    if strong_tag(point) == 1
                        onescount = onescount + 1;
                        if New_aggregatedSignal(point) >= temphigh
                            temphigh = New_aggregatedSignal(point);
                            highpointer = point; %pointer is for diagnostic
purposes
                        end
                        if New_aggregatedSignal(point) <= templow
                            templow = New_aggregatedSignal(point);
```

```
                            lowpointer = point; %pointer is for diagnostic
purposes
                        end
                    end
                end
            end
            %Calculate the estimate amplitude of the strongest tag
            addmaxmin = temphigh + templow;
            amplitude_i1 = (addmaxmin/2);
            %amplitude_i1 is the estimate of the strongest tag's Rx signal.

            %The section below provides some debugging diagnostics. The
            % variable tau_fail will count the number of times in the
            % entire simulation run that the estimate for tau was in error,
            % and the variable spread_overpowered will show the number of
            % times in the entire simulation run that the extracted data
            % from a tag was in error

            strong_tau_guess = (amplitude_i1/5); %This is the estimate of
            % the strongest tag's tau
            strong_tau = tau(guess_winner(1)); %This is the actual value
            % of tau for the strongest tag
            diagnosis(iruncount,1) = iruncount;
            diagnosis(iruncount,2) = strong_tau_guess;
            diagnosis(iruncount,3) = strong_tau;

            if abs(strong_tau - strong_tau_guess) > 0.001
                tau_fail = tau_fail+1;
                diagnosis(iruncount,5) = tau_fail;
            else
                diagnosis(iruncount,5) = 0;
            end
            break
        end  %This is the end of the (for n_aic) loop

        %Check if all potentially strongest tags have been tried and have
        % failed. If so, indicate that spreading code has been over_powered
        %and go to the end of the modulation loop
        flag=0;
        if datacheck == 1
            if n_aic >= n_strong
                flag =1;
                spread_overpowered = spread_overpowered + 1;
            end
        end
        if flag == 1
            break
        end

       %Eight step: subtract effects of strongest tag from received signal.
       % This step is the actual AIC (cancellation of the effects
       %of the strongest tag)
        for p = 1:no_ofbits
            for m = 1:processing_gain
                point = m + (p-1)*processing_gain;
```

```
                calculate(point) = New_aggregatedSignal(point)-((-1 +
2*strong_tag(point))*amplitude_i1);
            end
        end
        New_aggregatedSignal = calculate; % Storing the values in
        % New_aggregatedSignal now represents the received signal after the
        % effects of the strongest tag have been subtracted. We're now
        % ready to repeat the demodulation loop to extract data from
        % another tag.

    end %This is the end of the demodulation loop
end

successful_demodulation;
percent_messages_successfully_demodulated =
successful_demodulation/(no_oftags);

%--------------------------------------------------------------------------
% THIS PIECE OF CODE OUTPUT THE PERCENTAGE OF SUCCESSFULL DEMODULATED
% MESSAGES MULTIPLIED BY 100
%--------------------------------------------------------------------------
perc = percent_messages_successfully_demodulated;
fprintf('\n');

%--------------------------------------------------------------------------
% THIS SECTION SHOWS THE NUMBER OF DEMODULATED TAGS DEPENDING ON THE
% PERCENTAGE OF SUCCESFUL AND UNSUCCESFUL DEMODULATED MESSAGES
%--------------------------------------------------------------------------
fout = floor(perc*100);
fout1 = fout;
calc = floor((no_oftags*fout)/100);
ncalc = calc;

no_dem = no_oftags - ncalc;
no_dem2 = no_dem;

toc;
fprintf('\n');
```

Appendix D: CDMA/AIC Function 2 (Half processing gain)

```
%-------------------------------------------------------------------------
% FUNCTION 2 CDMA PROGRAM - SAME GP
% TAKEN FROM REFERENCE [8]
%-------------------------------------------------------------------------
%Parameters from previous run
no_oftags=no_dem2;
no_ofbits=256;
sim_runs=sims;
processing_gain=processing_gain/2;

%-------------------------------------------------------------------------
% NOTE THAT THE ELAPSED TIME STARTS HERE, THE TIME THAT THE USER SPEND
% INPUTING VARIABLES DOES NOT COUNT.
%-------------------------------------------------------------------------
tic; %tic-toc function to measure elapsed time

tau_fail=0; %Diagnostic variable
spread_overpowered=0; %Diagnostic variable
successful_demodulation=0; % Counter for number of tag messages
noise_power=1.577;

%The loop below(using the variable iruncount) spans most of the program
for iruncount=1:1
    tag_arr = randi(0:1, no_oftags, no_ofbits); %Generate array containing
    % original binary data for all tags
    scm = hadamard(processing_gain); %NO NEED TO TRANSPOSE THE MATRIX
    %The rows of scm are now orthogonal spreading codes
    %The next loop of 6 lines randomizes (or "scrambles") the rows of scm,
    % which helps remove correlation between adjacent rows and will
    % subsequently simplify the process of having each tag randomly choose
    % a spreading code
    for j=1:no_oftags
        scramble=randi(processing_gain); %NO_OFTAGS TIMES GET RANDOM
        % NUMBERS BETWEEN 1 AND 64

        %For no_oftags > Gp
        if j>processing_gain
            jm = mod (j,processing_gain)+1;
        else
            jm=j;
        end
        temp=scm(jm,:); %TAKES THE FIRST ROWN OF 64 BITS

        scm(j,:)=scm(scramble,:);
        scm(scramble,:)=temp; %SCM IS THE NEW MATRIX WHITH SCRAMBLED ROWS
    end

    sc_t=scm;
    sc_t(sc_t == -1)=0; % sc_t now contains the different spreading codes
    % in its rows, converted to 1s and 0s instead of 1s and -1s
    %The loops below create the matrix out_array. Each row of out_array
```

```
% contains the spread data corresponding to one of the tags.
%Because the rows of scm (and therefore sc_t) have been "scrambled",
% the process simulates each tag randomly selecting a spreading code.
out_array=zeros(no_oftags,processing_gain*no_ofbits);

for p=1:no_ofbits   %P GOES FROM 1 TO 256
    for n=1:no_oftags %n GOES FROM 1 TO 9
        for m =1:processing_gain %m GOES FROM 1 TO 64
            point=m+(p-1)*processing_gain;
            out_array(n,point)=xor(tag_arr(n,p),sc_t(n,m)); %Where the
            % spreading code is done
        end
    end
end

%The next section produces a Rayleigh distribution for the relative
% amplitude of each tag's transmission.
%This simulates the effect of multipath and shadowing
tau=zeros(1,no_oftags);
sigma=1;
for n=1:no_oftags
    tau(1,n)= sigma*sqrt(-2*log(1-rand(1))); %Numerical explanation in
    % separate page
end


%The next section creates the analog transmitted signal for each tag
%Nominally, 5 millivolts is used to represent a "1" and -5 millivolts
% is used to represent a "0", but each tag's signal must then be
%multiplied by the "tau" to include the effects of multipath and
% fading. Each row of the matrix volts_forall will contain the analog
%signal corresponding to one tag.
%This signals are PAM with rectangular pulses. We may need to modify
% them. Avg signal power = (5 millivolts)^2
volts_forall=zeros(no_oftags, no_ofbits*processing_gain);
for n = 1:no_oftags
    volts_forall(n,:)=(-5 + 10*out_array(n,:))*tau(1,n);
end
%Since all the signals are transmitted simultaneously, the total
% transmitted signal is the sum of all the individual signals. The
% section below creates analog_signal, which is the aggregate
% transmitted signal

volts_add=zeros(1,point);
for n = 1:no_oftags
    volts_add(1,:)=volts_add + (volts_forall(n,:));
end
transmitted_signal=volts_add; %Total signal seat over the channel

%Now add the noise to the analog transmitted signal. Channel
% attenuetion could be added, too, but won't change the analysis as
%long as received SNR is the parameter used to evaluate performance
noise_db = 10*log10(noise_power); %remember that signal and noise power
% measurements are given in millivolts^2
x_noise=wgn(point,1,noise_db); %WHITE GAUSSIAN NOISE FUNCTION -
```

```
    % VERTICAL VECTOR - LENGTH=16384, 1 ROW
    x_noise=transpose(x_noise);
    received_signal=transmitted_signal + x_noise; %Received signal has not
    % been attenuated (gamma=1) but noise had been added.

    %Now start demodulation,either of the received signal when (z=1) or of
    % the received signal after an AIC loop (when z>1)

    New_aggregatedSignal=received_signal; %Before AIC is applied
    % New_aggregatedSignal will be the same as the received signal
    %but not after AIC is applied.


    %The large loop below (using the variable z) demodulates, despreads,
    % finds the tag with the strongest signal, extracts the data
    %from the tag and then uses AIC to remove the effect of the strongest
    % tag
    for z=1:no_oftags
        %First step: demodulate the received signal. Later we may want to
        % use different variable to represent the analog received
        %signal and the demodulated received signal
        voltsadd_val(1:point)=(New_aggregatedSignal(1:point) +
abs(New_aggregatedSignal(1:point)))/2;
        voltsadd_val(voltsadd_val>0) = 1; %voltsadd_val is now the
        % demodulated, spread signal at the receiver (1s and 0s)

        %Second step: despreads the first bit of the received signal using
        % each possible spreading code (we are only dispreading the first
        % chip because that information will be sufficient to tell us which
        % tag sent the strongest signal)
        %The first "chip" is the first 64 bits of the of the spread signal.
        % After spreading, it will correspond to one bit of information
        add_signal = voltsadd_val(1:processing_gain); %add_signal is the
        % first chip of received signal
%-------------------------------------------------------------------------
% HERE THE PARAMETER OF "ZEROS" MATRIX WAS CHANGED, WE DO NOT NEED A 64X64
% MATRIX ANYMORE, THE #TAGS WILL DETERMINE THE NUMBER OF ROWS
%-------------------------------------------------------------------------
        despreading_forall = zeros (no_oftags, processing_gain);

        for n = 1:no_oftags
            despreading_forall(n,:) = xor(add_signal, sc_t(n,:));
        end
        %Each row of despreading_forall now contains the first chip of the
        % received signal xor-ed with one of the possible spreading codes
        despread_results=despreading_forall;

        %Third step: determine which spreading code produced the chip that
        % is most consistent (i.e., has the most 1s or 0s). That the code
        %will correspond to the strongest tag.
        counting_rows = zeros(no_oftags,2);
        for n = 1:no_oftags
            counting_rows(n,1) = nnz(despread_results(n,:) == 1);
            counting_rows(n,2) = nnz(despread_results(n,:) == 0);
        end
```

```
        %Each row of counting_rows contains consistency information for a
        % particular spreading code. The first element in the row contains
        %the number of 1s, the second element contains the number of 0s
        %Now identify the spread tags with the greatest consistency.
        subtract_1 = zeros(no_oftags,1);
        for n = 1:no_oftags
            subtract_1(n,1) = abs(counting_rows(n,1)- counting_rows(n,2));
        end
        highest_num = max(subtract_1); %higest_num= maximum differential of
        % 1s and 0s. The higher this value, the greater the consistency
        guess_winner = find(subtract_1 == highest_num);
        %guess_winner is a 1-column array containing the numbers of all
        % tags producing the greatest consistency (i.e., the strongest
        %tags). The first element in this array will be used as the
        % strongest tag.
        %Note that there may be multiple tags with the same, greatest
        % consistency.

        %In most cases, selecting any one of these tags for our first pass
        % through AIC will allow us to successfully extract the tags data.
        %However, if an error occurs we want to be able to try again using
        % each of the other "greatest consistency" tags to see if we can
        %extract that tag's data without error
        sz = size(guess_winner);
        n_strong=sz(1); %n_strong is the number of tags with the
        % greatest consistency
        %Create a loop for strongest tag
        for n_aic = 1:n_strong
            guess_winner_despread_code =
despread_results(guess_winner(n_aic),:);
            %guess_winner_despread_code is the despreads code corresponding
            % to the potentially strongest tag (i.e., one of the tags with
            %the greatest consistency)
            guess_winner_spreading_code = sc_t(guess_winner(n_aic),:);
            %guess_winner_despread_code is the spreading code corresponding
            % to the potentially strongest tag

            %Fourth step: Now that a potentially strongest tag has been
            % identified, despreads all the received data using only the
            %spreading code from the potentially strongest tag.
            for p = 1:no_ofbits
                for m = 1:processing_gain
                    point = m +(p-1)*processing_gain;
                    despread_strong_tag(point) =
xor(voltsadd_val(point),guess_winner_spreading_code(m));
                end
            end

            %Fifth step: extract the original unspread data from the
            % potentially strongest tag
            for p = 1:no_ofbits
                counting_ones=0;
                counting_zeros=0;
                for m = 1:processing_gain
```

```
                point = m+(p-1)*processing_gain;
                if despread_strong_tag(point) == 1
                    counting_ones = counting_ones + 1;
                else
                    counting_zeros = counting_zeros + 1;
                end
                if counting_ones >= counting_zeros
                    data(p) = 1;
                else
                    data(p) = 0;
                end
            end
        end %The array data now contains the extracted data from the
        % potentially strongest tag.

        %Sixth step: Verify that the extracted data is correct. In
        % practical applications this verification will be done using a
        %small Cyclic Redundancy Check (CRC) code. Since the CRC will
        % be necessary, whether the system uses conventional slotted
        %ALOHA or CDMA, it's easier in this simulation to just check
        % the extracted data against the original data. This shortcut
        %won't change the performance comparison of the slotted ALOHA
        % system versus CDMA.

        %If extracted data is correct, the code below will set
        % datacheck will equal 0. If the extracted data has one or more
        %errors, datacheck will be equal to 1.


%-------------------------------------------------------------------------
% THIS IS THE COMPARISON OF THE ORIGINAL AND DEMODULATED MESSAGE
%-------------------------------------------------------------------------
%This approach is compares bit-by-bit the elements of both arrays.
        datacheck=0;
        cc=1:no_ofbits; % CC IS A NEW VARIABLE IN THE CODE THAT
        % REPLACES THE FOR LOOP, IT TAKES VALUES FROM 1 TO 256

        y=tag_arr(guess_winner(n_aic),cc);
        x=data(cc);
        if x == y
            datacheck=0;
            %'Successful demodulation'
        else
            datacheck=1;
            %'Error in demodulation'
        end

        if datacheck == 1
            continue  %if extracted data has an error, go to the end
            % of the loop and start over with another potentially
            %strongest tag
        end
       successful_demodulation = successful_demodulation + 1;

        %Seventh step: estimate the amplitude of the received signal
```

```
            % corresponding only to the strongest tag.

            %First, recreate the spread data corresponding to the
            % strongest tag
            for p = 1:no_ofbits
                for m = 1:processing_gain
                    point = m + (p-1)*processing_gain;
                    strong_tag(point) = xor(data(p),
guess_winner_spreading_code(m));
                end
            end

            %Second, estimate the amplitude of the received signal
            % corresponding only to the strongest tag. The first part of
            % the estimate involves determining the max and min values of
            % the received signal for those bits where strong_tag=1. We
            % start by initializing some variables.

            %The variable temphigh is initialized to -100 instead of zero,
            % because in rare cases all appropriate values of v may be
            %negative but they will not all be less than -100. templow is
            % initiated to 100 instead of 0 because in rare cases all
            %appropriate values of v may be positive but they will not be
            % grater than 100. Later we may want to refine this code.

                %If messages are long enough, it's reasonable to assume
                % that sometime during transmission the strongest tag is
                %transmitting a "1" and all other tags are also
                % transmitting a "1" (high value), and that at some other
                %time the strongest tag is transmitting a "1" but all other
                % tags are transmitting a "0" (low value). The amplitude of
                %the strongest signal is ("high"+"low")/2

            onescount=0;
            temphigh=0;
            templow=0;
            highpointer=-100;
            lowpointer=100;

            for p = 1:no_ofbits
                for m = 1:processing_gain
                    point = m + (p-1)*processing_gain;

                    if strong_tag(point) == 1
                        onescount = onescount + 1;
                        if New_aggregatedSignal(point) >= temphigh
                            temphigh = New_aggregatedSignal(point);
                            highpointer = point; %pointer is for diagnostic
                            % purposes
                        end
                        if New_aggregatedSignal(point) <= templow
                            templow = New_aggregatedSignal(point);
                            lowpointer = point; %pointer is for diagnostic
                            % purposes
                        end
```

```
                end
            end
        end
    %Now we can calculate the estimate amplitude of the strongest tag
        addmaxmin = temphigh + templow;
        amplitude_i1 = (addmaxmin/2);
    %amplitude_i1 is the estimate of the strongest tag's received signal.

        %The section below provides some debugging diagnostics. The
        % variable tau_fail will count the number of times in the
        % entire simulation run that the estimate for tau was in error,
        % and the variable spread_overpowered will show the number of
        % times in the entire simulation run that the extracted data
        % from a tag was in error

        strong_tau_guess = (amplitude_i1/5); %This is the estimate of
        % the strongest tag's tau
        strong_tau = tau(guess_winner(1)); %This is the actual value of
        % tau for the strongest tag
        diagnosis(iruncount,1) = iruncount;
        diagnosis(iruncount,2) = strong_tau_guess;
        diagnosis(iruncount,3) = strong_tau;

        if abs(strong_tau - strong_tau_guess) > 0.001
            tau_fail = tau_fail+1;
            diagnosis(iruncount,5) = tau_fail;
        else
            diagnosis(iruncount,5) = 0;
        end
        break
  end  %This is the end of the (for n_aic) loop

   %Check if all potentially strongest tags have been tried and have
   % failed. If so, indicate that spreading code has been over_powered
   %and go to the end of the modulation loop
    flag=0;
    if datacheck == 1
        if n_aic >= n_strong
            flag =1;
            spread_overpowered = spread_overpowered + 1;
        end
    end
    if flag == 1
        break
    end

    %Eight step: subtract effects of strongest tag from received
    % signal. This step is the actual AIC (cancellation of the effects
    %of the strongest tag)
    for p = 1:no_ofbits
        for m = 1:processing_gain
            point = m + (p-1)*processing_gain;
            calculate(point) = New_aggregatedSignal(point)-((-1 +
2*strong_tag(point))*amplitude_i1);
        end
```

```
        end
        New_aggregatedSignal = calculate; % Storing the values in
        % New_aggregatedSignal now represents the received signal after
        % the effects of the strongest tag have been subtracted. We're now
        % ready to repeat the demodulation loop to extract data from another
tag.

    end %This is the end of the demodulation loop
end

successful_demodulation;
percent_messages_successfully_demodulated =
successful_demodulation/(no_oftags);

%-------------------------------------------------------------------------
------------------
% THIS PIECE OF CODE OUTPUT THE PERCENTAGE OF SUCCESSFULL DEMODULATED
%-------------------------------------------------------------------------
------------------
perc = percent_messages_successfully_demodulated;
% fprintf('\n');

%-------------------------------------------------------------------------
------------------
% THIS PART SHOWS THE NUMBER OF DEMODULATED TAGS DEPENDING ON THE PERCENTAGE
OF SUCCESFUL AND
% UNSUCCESFUL DEMODULATED MESSAGES
%-------------------------------------------------------------------------
------------------
fout = floor(perc*100);
fout1 = fout;
calc = floor((no_oftags*fout)/100);
ncalc = calc;

no_dem = no_oftags - ncalc;
no_dem2 = no_dem;

toc;
fprintf('\n');
```

Appendix E: CDMA/AIC Function 3 (Same processing gain)

```
%-------------------------------------------------------------------------
% FUNCTION 3 CDMA PROGRAM - SAME GP
% TAKEN FROM REFERENCE [8]
%-------------------------------------------------------------------------
%Parameters from previous run
no_oftags=no_dem2;
no_ofbits=256;
sim_runs=sims;
processing_gain=processing_gain;

%-------------------------------------------------------------------------
% NOTE THAT THE ELAPSED TIME STARTS HERE, THE TIME THAT THE USER SPEND
% INPUTING VARIABLES DOES NOT COUNT.
%-------------------------------------------------------------------------
tic; %tic-toc function to measure elapsed time

tau_fail=0; %Diagnostic variable
spread_overpowered=0; %Diagnostic variable
successful_demodulation=0; % Counter for number of tag messages
noise_power=1.577;

%The loop below(using the variable iruncount) spans most of the program

for iruncount=1:1
    tag_arr = randi(0:1, no_oftags, no_ofbits); %Generate array containing
    % original binary data for all tags
    scm = hadamard(processing_gain); %NO NEED TO TRANSPOSE THE MATRIX
    %The rows of scm are now orthogonal spreading codes
    %The next loop of 6 lines randomizes (or "scrambles") the rows of scm,
    % which helps remove correlation between adjacent rows and will
    % subsequently simplify the process of having each tag randomly choose
    % a spreading code
    for j=1:no_oftags
        scramble=randi(processing_gain); %NO_OFTAGS TIMES GET RANDOM
        % NUMBERS BETWEEN 1 AND 64

        %For no_oftags > Gp
        if j>processing_gain
            jm = mod (j,processing_gain)+1;
        else
            jm=j;
        end
        temp=scm(jm,:); %TAKES THE FIRST ROWN OF 64 BITS

        scm(j,:)=scm(scramble,:);
        scm(scramble,:)=temp; %SCM IS THE NEW MATRIX WHITH SCRAMBLED ROWS
    end

    sc_t=scm;
    sc_t(sc_t == -1)=0; % sc_t now contains the different spreading codes
    % in its rows, converted to 1s and 0s instead of 1s and -1s
```

```
%The loops below create the matrix out_array. Each row of out_array
% contains the spread data corresponding to one of the tags.
%Because the rows of scm (and therefore sc_t) have been "scrambled",
% the process simulates each tag randomly selecting a spreading code.
out_array=zeros(no_oftags,processing_gain*no_ofbits);

for p=1:no_ofbits   %P GOES FROM 1 TO 256
    for n=1:no_oftags %n GOES FROM 1 TO 9
        for m =1:processing_gain %m GOES FROM 1 TO 64
            point=m+(p-1)*processing_gain;
            out_array(n,point)=xor(tag_arr(n,p),sc_t(n,m)); %Where the
            % spreading code is done
        end
    end
end

%The next section produces a Rayleigh distribution for the relative
% amplitude of each tag's transmission.
%This simulates the effect of multipath and shadowing
tau=zeros(1,no_oftags);
sigma=1;
for n=1:no_oftags
    tau(1,n)= sigma*sqrt(-2*log(1-rand(1))); %Numerical explanation
    % in separate page
end


%The next section creates the analog transmitted signal for each tag
%Nominally, 5 millivolts is used to represent a "1" and -5 millivolts
% is used to represent a "0", but each tag's signal must then be
%multiplied by the "tau" to include the effects of multipath and
% fading. Each row of the matrix volts_forall will contain the analog
%signal corresponding to one tag.
    %This signals are PAM with rectangular pulses. We may need to
    % modify them. Avg signal power = (5 millivolts)^2
volts_forall=zeros(no_oftags, no_ofbits*processing_gain);
for n = 1:no_oftags
    volts_forall(n,:)=(-5 + 10*out_array(n,:))*tau(1,n);
end
%Since all the signals are transmitted simultaneously, the total
% transmitted signal is the sum of all the individual signals. The
% section below creates analog_signal, which is the aggregate
% transmitted signal

volts_add=zeros(1,point);
for n = 1:no_oftags
    volts_add(1,:)=volts_add + (volts_forall(n,:));
end
transmitted_signal=volts_add; %Total signal seat over the channel

%Now add the noise to the analog transmitted signal. Channel
% attenuation could be added, too, but won't change the analysis as
%long as received SNR is the parameter used to evaluate performance
noise_db = 10*log10(noise_power); %remember that signal and noise power
% measurements are given in millivolts^2
```

```
    x_noise=wgn(point,1,noise_db); %WHITE GAUSSIAN NOISE FUNCTION -
    % VERTICAL VECTOR - LENGTH=16384, 1 ROW
    x_noise=transpose(x_noise);
    received_signal=transmitted_signal + x_noise; %Received signal has not
    % been attenuated (gamma=1) but noise had been added.

    %Now start demodulation,either of the received signal when (z=1) or of
    % the received signal after an AIC loop (when z>1)

    New_aggregatedSignal=received_signal; %Before AIC is applied
    % New_aggregatedSignal will be the same as the received signal
    %but not after AIC is applied.


    %The large loop below (using the variable z) demodulates, despreads,
    % finds the tag with the strongest signal, extracts the data
    %from the tag and then uses AIC to remove the effect of the strongest
    % tag
    for z=1:no_oftags
        %First step: demodulate the received signal. Later we may want to
        % use different variable to represent the analog received
        %signal and the demodulated received signal
        voltsadd_val(1:point)=(New_aggregatedSignal(1:point) +
abs(New_aggregatedSignal(1:point)))/2;
        voltsadd_val(voltsadd_val>0) = 1; %voltsadd_val is now the
        % demodulated, spread signal at the receiver (1s and 0s)

        %Second step: despreads the first bit of the received signal using
        % each possible spreading code (we are only
        %dispreading the first chip because that information will be
        % sufficient to tell us which tag sent the strongest signal)
            %The first "chip" is the first 64 bits of the of the spread
            % signal. After spreading, it will correspond to one bit of
            %information
        add_signal = voltsadd_val(1:processing_gain); %add_signal is the
        % first chip of received signal
%-------------------------------------------------------------------------
% HERE THE PARAMETER OF "ZEROS" MATRIX WAS CHANGED, WE DO NOT NEED A 64X64
% MATRIX ANYMORE, THE #TAGS WILL DETERMINE THE NUMBER OF ROWS
%-------------------------------------------------------------------------
        despreading_forall = zeros (no_oftags, processing_gain);

        for n = 1:no_oftags
            despreading_forall(n,:) = xor(add_signal, sc_t(n,:));
        end
        %Each row of despreading_forall now contains the first chip of the
        % received signal xor-ed with one of the possible spreading codes
        despread_results=despreading_forall;

        %Third step: determine which spreading code produced the chip that
        % is most consistent (i.e., has the most 1s or 0s). That the code
        %will correspond to the strongest tag.
        counting_rows = zeros(no_oftags,2);
        for n = 1:no_oftags
            counting_rows(n,1) = nnz(despread_results(n,:) == 1);
```

```
            counting_rows(n,2) = nnz(despread_results(n,:) == 0);
        end


        %Each row of counting_rows contains consistency information for a
        % particular spreading code. The first element in the row contains
        %the number of 1s, the second element contains the number of 0s
        %Now identify the spread tags with the greatest consistency.
        subtract_1 = zeros(no_oftags,1);
        for n = 1:no_oftags
            subtract_1(n,1) = abs(counting_rows(n,1)- counting_rows(n,2));
        end
        highest_num = max(subtract_1); %higest_num= maximum differential of
        % 1s and 0s. The higher this value, the greater the consistency
        guess_winner = find(subtract_1 == highest_num);
        %guess_winner is a 1-column array containing the numbers of all
        % tags producing the greatest consistency (i.e., the strongest
        %tags). The first element in this array will be used as the
        % strongest tag.
        %Note that there may be multiple tags with the same, greatest
        % consistency.

        %In most cases, selecting any one of these tags for our first pass
        % through AIC will allow us to successfully extract the tags data.
        %However, if an error occurs we want to be able to try again using
        % each of the other "greatest consistency" tags to see if we can
        %extract that tag's data without error
        sz = size(guess_winner);
        n_strong=sz(1); %n_strong is the number of tags with the greatest
        % consistency
        %Create a loop for strongest tag
        for n_aic = 1:n_strong
            guess_winner_despread_code =
despread_results(guess_winner(n_aic),:);
            %guess_winner_despread_code is the despreads code corresponding
            % to the potentially strongest tag (i.e., one of the tags with
            %the greatest consistency)
            guess_winner_spreading_code = sc_t(guess_winner(n_aic),:);
            %guess_winner_despread_code is the spreading code corresponding
            %to the potentially strongest tag

            %Fourth step: Now that a potentially strongest tag has been
            % identified, despreads all the received data using only the
            %spreading code from the potentially strongest tag.
            for p = 1:no_ofbits
                for m = 1:processing_gain
                    point = m +(p-1)*processing_gain;
                    despread_strong_tag(point) =
xor(voltsadd_val(point),guess_winner_spreading_code(m));
                end
            end

            %Fifth step: extract the original unspread data from the
            % potentially strongest tag
            for p = 1:no_ofbits
```

```
                counting_ones=0;
                counting_zeros=0;
                for m = 1:processing_gain
                    point = m+(p-1)*processing_gain;
                    if despread_strong_tag(point) == 1
                        counting_ones = counting_ones + 1;
                    else
                        counting_zeros = counting_zeros + 1;
                    end
                    if counting_ones >= counting_zeros
                        data(p) = 1;
                    else
                        data(p) = 0;
                    end
                end
            end %The array data now contains the extracted data from the
            % potentially strongest tag.

            %Sixth step: Verify that the extracted data is correct. In
            % practical applications this verification will be done using a
            %small Cyclic Redundancy Check (CRC) code. Since the CRC will
            % be necessary, whether the system uses conventional slotted
            %ALOHA or CDMA, it's easier in this simulation to just check
            % the extracted data against the original data. This shortcut
            %won't change the performance comparison of the slotted ALOHA
            % system versus CDMA.

            %If extracted data is correct, the code below will set
            % datacheck will equal 0. If the extracted data has one or more
            %errors, datacheck will be equal to 1.

%-------------------------------------------------------------------------
% THIS IS THE COMPARISON OF THE ORIGINAL AND DEMODULATED MESSAGE
%-------------------------------------------------------------------------
            %This approach is COMPARES BIT BY BIT THE ELEMENTS OF BOTH
            % ARRAYS.
            datacheck=0;
            cc=1:no_ofbits; % CC IS A NEW VARIABLE IN THE CODE THAT
            % REPLACES THE FOR LOOP, IT TAKES VALUES FROM 1 TO 256

            y=tag_arr(guess_winner(n_aic),cc);
            x=data(cc);
            if x == y
                datacheck=0;
                %'Successful demodulation'
            else
                datacheck=1;
                %'Error in demodulation'
            end


            if datacheck == 1
                continue  %if extracted data has an error, go to the end
                % of the loop and start over with another potentially
                %strongest tag
```

```
            end
           successful_demodulation = successful_demodulation + 1;


            %Seventh step: estimate the amplitude of the received signal
            % corresponding only to the strongest tag.

            %First, recreate the spread data corresponding to the strongest
            % tag
            for p = 1:no_ofbits
                for m = 1:processing_gain
                    point = m + (p-1)*processing_gain;
                    strong_tag(point) = xor(data(p),
guess_winner_spreading_code(m));
                end
            end

            %Second, estimate the amplitude of the received signal
            % corresponding only to the strongest tag. The first part of
            % the estimate involves determining the max and min values of
            % the received signal for those bits where strong_tag=1.
            % We start by initializing some variables.

            %The variable temphigh is initialized to -100 instead of zero,
            % because in rare cases all appropriate values of v may be
            %negative but they will not all be less than -100. templow is
            % initiated to 100 instead of 0 because in rare cases all
            %appropriate values of v may be positive but they will not be
            % grater than 100. Later we may want to refine this code.

                %If messages are long enough, it's reasonable to assume
                % that sometime during transmission the strongest tag is
                %transmitting a "1" and all other tags are also
                % transmitting a "1" (high value), and that at some other
                %time the strongest tag is transmitting a "1" but all other
                % tags are transmitting a "0" (low value). The amplitude of
                %the strongest signal is ("high"+"low")/2

            onescount=0;
            temphigh=0;
            templow=0;
            highpointer=-100;
            lowpointer=100;

            for p = 1:no_ofbits
                for m = 1:processing_gain
                    point = m + (p-1)*processing_gain;

                    if strong_tag(point) == 1
                        onescount = onescount + 1;
                        if New_aggregatedSignal(point) >= temphigh
                            temphigh = New_aggregatedSignal(point);
                            highpointer = point; %pointer is for diagnostic
purposes
                        end
```

```
                        if New_aggregatedSignal(point) <= templow
                            templow = New_aggregatedSignal(point);
                            lowpointer = point; %pointer is for diagnostic
purposes
                        end
                    end
                end
            end
            %Calculate the estimate amplitude of the strongest tag
            addmaxmin = temphigh + templow;
            amplitude_i1 = (addmaxmin/2);
            %amplitude_i1 is the estimate of the strongest tag's Rx signal.

            %The section below provides some debugging diagnostics. The
            % variable tau_fail will count the number of times in the
            % entire simulation run that the estimate for tau was in error,
            % and the variable spread_overpowered will show the number of
            % times in the entire simulation run that the extracted data
            % from a tag was in error

            strong_tau_guess = (amplitude_i1/5); %This is the estimate of
            % the strongest tag's tau
            strong_tau = tau(guess_winner(1)); %This is the actual value
            % of tau for the strongest tag
            diagnosis(iruncount,1) = iruncount;
            diagnosis(iruncount,2) = strong_tau_guess;
            diagnosis(iruncount,3) = strong_tau;

            if abs(strong_tau - strong_tau_guess) > 0.001
                tau_fail = tau_fail+1;
                diagnosis(iruncount,5) = tau_fail;
            else
                diagnosis(iruncount,5) = 0;
            end
            break
        end  %This is the end of the (for n_aic) loop

        %Check if all potentially strongest tags have been tried and have
        %failed. If so, indicate that spreading code has been over_powered
        %and go to the end of the modulation loop
        flag=0;
        if datacheck == 1
            if n_aic >= n_strong
                flag =1;
                spread_overpowered = spread_overpowered + 1;
            end
        end
        if flag == 1
            break
        end

        %Eight step: suftract effects of strongest tag from Rx signal. This
        % step is the actual AIC (cancellation of the effects
        %of the strongest tag)
        for p = 1:no_ofbits
```

```
            for m = 1:processing_gain
                point = m + (p-1)*processing_gain;
                calculate(point) = New_aggregatedSignal(point)-((-1 +
2*strong_tag(point))*amplitude_i1);
            end
        end
        New_aggregatedSignal = calculate; % Storing the values in
        % New_aggregatedSignal now represents the received signal after
        % the effects of the strongest tag have been subtracted. We're now
        % ready to repeat the demodulation loop to extract data from
        % another tag.

    end %This is the end of the demodulation loop
end

successful_demodulation;
percent_messages_successfully_demodulated =
successful_demodulation/(no_oftags);

%--------------------------------------------------------------------------
% THIS PIECE OF CODE OUTPUT THE PERCENTAGE OF SUCCESSFULL DEMODULATED
% MESSAGES MULTIPLIED BY 100
%--------------------------------------------------------------------------
perc = percent_messages_successfully_demodulated;
fprintf('\n');

%--------------------------------------------------------------------------
% THIS PART SHOWS THE NUMBER OF DEMODULATED TAGS DEPENDING ON THE
% PERCENTAGE OF SUCCESFUL AND
% UNSUCCESFUL DEMODULATED MESSAGES
%--------------------------------------------------------------------------
fout = floor(perc*100);
fout1 = fout;
calc = floor((no_oftags*fout)/100);
ncalc = calc;

no_dem = no_oftags - ncalc;
no_dem2 = no_dem;

toc;
fprintf('\n');

ratio = 100*ncalc/(ncalc+no_dem2);
ratiostr=ratio;
```

Appendix F: DFSA main code

```
%--------------------------------------------------------------------------
% MAIN DFSA PROGRAM
%--------------------------------------------------------------------------
%Before starting all the registers will be cleaned up.
clear
clc

sigma=1;
no_oftags=input('Input the number of tags:   ');
no_oftags2=no_oftags;
no_ofbits = input('Input the number of bits:   ');
no_ofslots = input('Input the number of slots:   ');
sim_runs = input('How many times the program will run?   ');
pointr=0; % Pointer for building the super matrix
fprintf('\n');
fprintf('------------------------------------------------------------------------');
fprintf('\n');

%--------------------------------------------------------------------------
% RESET OF THE VARIABLES
% The following variables keep the value of the original input values of
% the variables to work with them in the different runs
%--------------------------------------------------------------------------
reset_tags = no_oftags;
reset_slot = no_ofslots;

%--------------------------------------------------------------------------
%MULTIPLE SIMULATIONS LOOP
%--------------------------------------------------------------------------
for n=1:sim_runs
    simcount = n;
    no_oftags = reset_tags;
    missed_tags=reset_tags;
    slot=reset_slot;
    sims = sim_runs;


%--------------------------------------------------------------------------
%INITIALIZATION OF THE VARIABLES FOR THE OUTPUT MATRIX
%--------------------------------------------------------------------------
iterationUntilCompletion=1; %INITIALIZATION OF THE COUNTER FOR THE
% ITERATIONS
iter_change=zeros(1,1000); % MATRIX OF ZEROS FOR STORING THE VALUES OF THE
% ITERATION WHILE THEY CHANGE

sim=0;
sim_change=zeros(1,1000); % MATRIX OF ZEROS FOR STORING THE VALUES OF THE
% NUMBER OF SIMULATIONS WHILE THEY CHANGE

i=0;
slots_change=zeros(1,1000); % MATRIX OF ZEROS FOR STORING THE VALUES OF THE
```

```
% SLOTS WHILE THEY CHANGE

fail=0;
fail_change=zeros(1,1000); % MATRIX OF ZEROS FOR STORING THE VALUES OF THE
% UNSUCCESSFUL DEMODULATED TAGS WHILE THEY CHANGE

success=0;
success_change=zeros(1,1000); % MATRIX OF ZEROS FOR STORING THE VALUES OF
% THE SUCCESSFUL DEMODULATED TAGS WHILE THEY CHANGE

%The elapsed time starts after the first test
tic

FSA_Same_V4
out = ['Percent of messages in error:    ', num2str(perc), '%'];
disp (out)
fprintf('\n');
out = ['Number of missed tags:    ', num2str(miss)];
disp (out)
fprintf('\n');
out = ['Number of successful demodulated tags:    ', num2str(successful)];
disp (out)
fprintf('\n');
out = ['The number of slots are the same:    ', num2str(sl)];
disp (out)
fprintf('\n');
fprintf('----------------------------------------------------------------');
pause(3)
slot = no_ofslots;
fprintf('\n');

% missed_tags is the variable that indicates the number of unsuccessful
% demodulated tags, meanwhile that variable is greater than 0, the loop
% keeps running.

while (missed_tags > 0)

    iterationUntilCompletion= iterationUntilCompletion + 1 ; %COUNTER FOR
    % THE ITERATIONS
    iter_change(iterationUntilCompletion)=iterationUntilCompletion;

    sim=sim+1; %COUNTER FOR THE NUMBER OF SIMULATIONS
    sim_change(sim)=sims;

    i=i+1; %COUNTER FOR CHANGES OF THE NUMBER OF SLOTS
    slots_change(i)=slot;

    fail=fail+1; %COUNTER FOR THE UNSUCCESSFUL DEMODULATED TAGS
    fail_change(fail)=missed_tags;

    success=success+1; %COUNTER FOR THE SUCCESSFUL DEMODULATED TAGS
    success_change(success)=success_tags;


% If the percentage of error (Perc_error) is between 0 and 40 that means
```

```
% that the most of the tags were demodulated, so the number of slots is
% reduced to the half
% If the percentage of error (Perc_error) is between 41 and 70 that means
% that a little bit more than the half of the tags were demodulated, so the
% number of slots is kept as the same
% If the percentage of error (Perc_error) is between  70 and 99 that means
% that a small number of tags were demodulated, so the number of slots is
% doubled

    if Perc_error >= 0 && Perc_error <= 40
        FSA_Half_V4 % Half slots
        slot = no_ofslots;
        out = ['Percent of messages in error:   ', num2str(perc), '%'];
        disp (out)
        fprintf('\n');
        out = ['Number of missed tags:   ', num2str(miss)];
        disp (out)
        fprintf('\n');
        out = ['Number of successful demodulated tags:   ',
num2str(successful)];
        disp (out)
        fprintf('\n');
        out = ['The number of slots were reduced to the half:   ',
num2str(sl)];
        disp (out)
        fprintf('\n');
        fprintf('-------------------------------------------------------------
----');
        pause (3)
        fprintf('\n');

    elseif Perc_error > 40 && Perc_error <= 70
        FSA_Same_V4 % Keep the slots
        slot = no_ofslots;
        out = ['Percent of messages in error:   ', num2str(perc), '%'];
        disp (out)
        fprintf('\n');
        out = ['Number of missed tags:   ', num2str(miss)];
        disp (out)
        fprintf('\n');
        out = ['Number of successful demodulated tags:   ',
num2str(successful)];
        disp (out)
        fprintf('\n');
        out = ['The number of slots are the same:   ', num2str(sl)];
        disp (out)
        fprintf('\n');
        fprintf('-------------------------------------------------------------
----');
        pause (3)
        fprintf('\n');

    else
        FSA_Double_V4 % Double slots
        slot = no_ofslots;
```

```
        out = ['Percent of messages in error:   ', num2str(perc), '%'];
        disp (out)
        fprintf('\n');
        out = ['Number of missed tags:   ', num2str(miss)];
        disp (out)
        fprintf('\n');
        out = ['Number of successful demodulated tags:   ',
num2str(successful)];
        disp (out)
        fprintf('\n');
        out = ['The number of slots were doubled:   ', num2str(sl)];
        disp (out)
        fprintf('\n');
        fprintf('-------------------------------------------------------------
----');
        pause (3)
        fprintf('\n');

    end
end

toc

change_sizeslot=iterationUntilCompletion; % KEEP THE VALUE AS A NUMBER
% BEFORE MAKING IT A STRING FOR DISPLAYING IT
fprintf('\n');
iteration_num = ['Number of times the slots changed their size: ',
num2str(change_sizeslot)];
disp (iteration_num)
fprintf('\n');

%-------------------------------------------------------------------------
% THE OUTPUT MATRIX SHOWS:
%   - FIRST COLUMN: SIMULATION NUMBER
%   - SECOND COLUMN: PARTICULAR ROUND
%   - THIRD COLUMN: NUMBER OF SLOTS
%   - FOURTH COLUMN: NUMBER OF UNSUCCESSFUL DEMODULATED TAGS
%   - FIFTH COLUMN: NUMBER OF SUCCESSFUL DEMODULATED TAGS
%-------------------------------------------------------------------------
matrix=zeros(1000,5); %MATRIX FOR THE OUTPUT VALUES:RUNS, ROUNDS, SLOTS,
% SUCCESS AND FAIL

%-------------------------------------------------------------------------
% IN THIS PIECE OF CODE IS SHOWN HOW THE VARIABLES WILL BE CHANGING IN
% EVERY ITERATION OF THE PROGRAM.
% VARIABLES: NUMBER OF RUN, ROUND, NUMBER OF SLOTS, UNSUCCESSFUL, AND
% SUCCESFUL DEMODULATED TAGS.
%-------------------------------------------------------------------------
iter_change(iterationUntilCompletion)=change_sizeslot;
iter_change(1,1)=1;
iter_change = transpose(iter_change);%SHOWS THE CHANGES OF THE SIMULATION
% NUMBER AS A VERTICAL VECTOR

sim_change(sim+1)=sim_runs;
sim_change(1,1)=1;
```

```
sim_change = transpose(sim_change);

slots_change(i+1)=slot;
slots_plot= slots_change; %Vector of number of slots to be plotted
slots_plot( :, ~any(slots_plot,1) ) = [];
slots_change = transpose(slots_change); % SHOWS THE CHANGES OF THE SLOTS AS
% A VERTICAL VECTOR

fail_change(fail+1)=missed_tags;
fail_change=transpose(fail_change); % SHOWS THE CHANGES OF THE UNSUCCESSFUL
% TAGS AS A VERTICAL VECTOR

success_change(success+1)=success_tags;
success_plot= success_change; %Vector of demodulated tags to be plotted
success_plot( :, ~any(success_plot,1) ) = [];
success_change=transpose(success_change); % SHOWS THE CHANGES OF THE
% UNSUCCESSFUL TAGS AS A VERTICAL VECTOR

fprintf('\n');
fprintf('------------------------------------------------------------------------');

%-------------------------------------------------------------------------
% PRINTING THE MATRIX PER RUN, EVERY ITERATING VARIABLE SHOWN IN A COLUM
% OF THE MATRIX
%-------------------------------------------------------------------------
matrix(:,1)=iter_change; %STORING THE VALUES OF THE SIMULATION NUMBER IN
% THE FIRST ROW OF THE MATRIX
matrix(:,2)=sim_change; %STORING THE VALUES OF THE ACTUAL RUN IN THE SECOND
% ROW OF THE MATRIX
matrix(:,3)=slots_change; %STORING THE VALUES OF THE SLOT IN THE THIRD ROW
% OF THE MATRIX
matrix(:,4)=fail_change; %STORING THE VALUES OF THE UNSUCCESSFUL
% DEMODULATED TAGS IN THE FOURTH ROW OF THE MATRIX
matrix(:,5)=success_change; %STORING THE VALUES OF THE SUCCESSFUL
% DEMODULATED TAGS IN THE FIFTH ROW OF THE MATRIX

%-------------------------------------------------------------------------
% WORKING ON THE SECOND COLUMN OF THE MATRIX
% Overwriting method
%-------------------------------------------------------------------------
for jx=1:change_sizeslot
    matrix(jx,2) = simcount;
end

%-------------------------------------------------------------------------
% SHOWS THE OUTPUT MATRIX AND GET RID OF THE ROWS THAT ARE ZEROS
%-------------------------------------------------------------------------
matrix;
matrix( ~any(matrix,2), : ) = []  %Get rid of the zero rows and shows the
% final matrix
fprintf('\n');
fprintf('------------------------------------------------------------------------');
fprintf('\n');

success_plot = transpose(matrix(:,5));
```

```
%-------------------------------------------------------------------------
% PLOT OF THE CHANGE OF THE NUMBER OF SLOTS VS NUMBER OF RUNS
%-------------------------------------------------------------------------
subplot(2,1,1);
plot(slots_plot,'r-o','LineWidth',1.5)
grid on
title('Change in the number of slots')
xlabel('Number of runs')
ylabel('Number of slots')
%-------------------------------------------------------------------------
% PLOT OF THE CHANGE OF THE SUCESSFUL DEMODULATED TAGS VS NUMBER OF RUNS
%-------------------------------------------------------------------------
subplot(2,1,2);
plot(success_plot,'b-o','LineWidth',1.5)
grid on
title('Successfully demodulated tags')
xlabel('Number of runs')
ylabel('Succesfully demodulated tags')

%-------------------------------------------------------------------------
% SUPERMATRIX
%-------------------------------------------------------------------------
for i=1:change_sizeslot
    for j=1:5
        supermatrix(i+pointr,j) = matrix(i,j);
    end
end

pointr = pointr + change_sizeslot;

end %end of the loop for each simulation

pause(5)
supermatrix

%-------------------------------------------------------------------------
% DISPLAY OF THE SUPER MATRIX
% This line of code export the data from Matlab to excel the argument of
% the function is:
%   - supermatrix.xlsx: the name of the excel file that has to be in the
% same folder as the programs
%   - supermatrix: the matrix to be printed in the excel file
%   - Sheet 2: sheet in which the data will be exported
%   - A2: cell since where the data will be printed
%-------------------------------------------------------------------------
xlswrite('supermatrix.xlsx', supermatrix, 'Sheet 2', 'A2')
```

Appendix G: DFSA Function 1 (Double size of frame)

```
%-------------------------------------------------------------------------
% FUNCTION 1 DFSA PROGRAM - DOUBLE SLOTS
% TAKEN FROM REFERENCE [8]
%-------------------------------------------------------------------------
%Parameters from previous run
no_ofslots=slot*2;
no_oftags=missed_tags;
no_oftagsnew=missed_tags;

max=1;
collisions=0;
message_errors=0;
total_rounds=0;
overflow=0;
missed_tags=0;
noise_power=2.058; % Scalar value of noise power (sigma squared) in mV^2.

%The loop below (using the variable iruncount) spans most of the program
% and runs the simulation "sim_runs" times.
for iruncount=1:1

  %If, after the previous round, there are either collisions or packets
  % received in error, then another round of transmissions will be
  % required for all the packets involved in collisions or errors.
  % The loop below reduces the number of tags to only those in collision
  % or error and  provides for another round of transmission.
    for irounds=1:max
        collisionflag = zeros(1,no_oftags);   %collisionflag is a 1xno_oftags
        % array that will be used to indicate whether or not a particular
        % tag's packet was involved in a collision
        errorflag = zeros(1,no_oftags);       %errorflag is a 1xno_oftags
        % array that will be used to indicate whether or not a particular
        % tag's packet was received in error.

 % Randomly choose a slot for each tag
        slotchoice = zeros(1,1000);
        for tag =1:no_oftags
            slotchoice(tag)=randi(no_ofslots); %vector 1Xno_oftags
        end

% This loop determines the number of messages that collide because their
% tags chose the same slot
       for j=1:no_oftags
            for k=j+1:no_oftags
                if slotchoice(j)==slotchoice(k)
                    collisionflag(j)=1;
                    collisionflag(k)=1;
                end
            end
        end
        collisions=collisions+sum(collisionflag);    % "collisions" contains
```

```
        % total number of collisions in all the simulation runs.  It's not
        % a very useful number except for diagnostics

% The loop below determines if an uncollided tag's message is successfully
% transmitted or if noise causes a packet error
    for j=1:no_oftags
            if collisionflag(j)==1
    continue % If the tab was involved in a collision there is no
    % reason to check noise
    else
    tau = sigma*sqrt(-2*log(1-rand(1)));      %add Rayleigh fading
    noise_db = 10*log10(noise_power);
                for k=1:no_ofbits
    if errorflag(j)==1 %skip loop if error has already been detected in tag

       break
                    end
                    bitval=randi(0:1);
                    x_noise = wgn(1,1,noise_db);
                    if bitval==0 % This IF statement checks to see if a
                   % 0 was transmitted and the noise was large and positive

                      if x_noise>=5*tau
                         errorflag(j)=1;
                      end
                    else  %This ELSE statement checks to see if a 1 was
                        % transmitted and the noise was large and negative
                      if x_noise<-5*tau
                        errorflag(j)=1;
                      end
                  end
              end
          end
        end
      no_oftags=sum(collisionflag)+sum(errorflag);  %Calculate the number
      % of tags that will need to be transmitted for the next round

    total_rounds=total_rounds+1;
        if no_oftags==0
            break
        end
        if irounds==max
            overflow=overflow+1;
            missed_tags=missed_tags+no_oftags;
            break
        end
    end
message_errors=message_errors+sum(errorflag); % "message_errors" contains
% total number of message errors due to noise

end

total_rounds;%The average number of rounds per simulation will be
% total_rounds/sim_runs
overflow;    %This is the number of times per sim_runs that there was at
```

```
% least one tag's data still in error after the maximum number of rounds

missed_tags; %It's possible that more than one tag was still in error
% after the maximum number of rounds.  missed_tags is the total number of
% tags that were not correctly received after sim_runs simulations.

%Calculation of the percentage of error in the process
Perc_error = (100*missed_tags)/no_oftagsnew;
no_ofslots;

perc = Perc_error;
miss = missed_tags;

success_tags=no_oftagsnew-missed_tags;
successful = success_tags;

sl = no_ofslots;
```

Appendix H: DFSA Function 2 (Half size of frame)

```
%-------------------------------------------------------------------------
% FUNCTION 2 DFSA PROGRAM - HALF SLOTS
% TAKEN FROM REFERENCE [8]
%-------------------------------------------------------------------------

%Condition to check on the slot size in the previous run, it will reduce
%the size of the slot if the previous one was 4 or greater, else it will
%keep it as 2
if slot >= 4
    no_ofslots=slot/2;
else
end

%Parameters from previous run
no_oftags=missed_tags;
no_oftagsnew=missed_tags;

max=1;
collisions=0;
message_errors=0;
total_rounds=0;
overflow=0;
missed_tags=0;
noise_power=2.058;% Scalar value of noise power (sigma squared) in mV^2.

%The loop below (using the variable iruncount) spans most of the program
% and runs the simulation "sim_runs" times.
for iruncount=1:1

  %If, after the previous round, there are either collisions or packets
  % received in error, then another round of transmissions will be required
  % for all the packets involved in collisions or errors.
  % The loop below reduces the number of tags to only those in collision
  % or error and  provides for another round of transmission.
    for irounds=1:max
        collisionflag = zeros(1,no_oftags);   %collisionflag is a 1xno_oftags
        % array that will be used to indicate whether or not a particular
        % tag's packet was involved in a collision
        errorflag = zeros(1,no_oftags); %errorflag is a 1xno_oftags array
        % that will be used to indicate whether or not a particular tag's
        % packet was received in error.

 % Randomly choose a slot for each tag
        slotchoice = zeros(1,1000);
        for tag =1:no_oftags
            slotchoice(tag)=randi(no_ofslots); %VECTOR 1Xno_oftags
        end

% This loop determines the number of messages that collide because their
% tags chose the same slot
       for j=1:no_oftags
```

```
            for k=j+1:no_oftags
                if slotchoice(j)==slotchoice(k);
                    collisionflag(j)=1;
                    collisionflag(k)=1;
                end
            end
        end
        collisions=collisions+sum(collisionflag);    % "collisions" contains
        % total number of collisions in all the simulation runs.  It's not
        % a very useful number except for diagnostics

% The loop below determines if an uncollided tag's message is successfully
% transmitted or if noise causes a packet error
    for j=1:no_oftags
            if collisionflag(j)==1
    continue % If the tab was involved in a collision there is no reason
    % to check noise
    else
    tau = sigma*sqrt(-2*log(1-rand(1)));      %add Rayleigh fading
    noise_db = 10*log10(noise_power);
                for k=1:no_ofbits
    if errorflag(j)==1%skip loop if error has already been detected in tag

       break
                    end
                    bitval=randi(0:1);
                    x_noise = wgn(1,1,noise_db);
                    if bitval==0 % This IF statement checks to see if a
                   % 0 was transmitted and the noise was large and positive

                      if x_noise>=5*tau
                         errorflag(j)=1;
                      end
                    else  %This ELSE statement checks to see if a 1 was
                        % transmitted and the noise was large and negative
                      if x_noise<-5*tau
                        errorflag(j)=1;
                      end
                   end
                end
            end
        end
      no_oftags=sum(collisionflag)+sum(errorflag);  %Calculate the number
      % of tags that will need to be transmitted for the next round

    total_rounds=total_rounds+1;
        if no_oftags==0
            break
        end
        if irounds==max
            overflow=overflow+1;
            missed_tags=missed_tags+no_oftags;
            break
        end
    end
```

```
message_errors=message_errors+sum(errorflag); % "message_errors" contains
% total number of message errors due to noise

end

total_rounds;%The average number of rounds per simulation will be total_
% rounds/sim_runs
overflow;    %This is the number of times per sim_runs that there was at
% least one tag's data still in error after the maximum number of rounds

missed_tags; %It's possible that more than one tag was still in error
% after the maximum number of rounds.  missed_tags is the total number of
% tags that were not correctly received after sim_runs simulations.

%Calculation of the percentage of error in the process
Perc_error = (100*missed_tags)/no_oftagsnew;
no_ofslots;

%Calculation of the percentage of error in the process
perc = Perc_error;
miss = missed_tags;

success_tags=no_oftagsnew-missed_tags;
successful = success_tags;

sl = no_ofslots;
```

Appendix I: DFSA Function 3 (Same size of frame)

```
%-------------------------------------------------------------------------
% FUNCTION 3 DFSA PROGRAM - SAME SLOTS
% TAKEN FROM REFERENCE [8]
%-------------------------------------------------------------------------
%Parameters from previous run
no_ofslots=slot;
no_oftags=missed_tags;
no_oftagsnew=missed_tags;

max=1;
collisions=0;
message_errors=0;
total_rounds=0;
overflow=0;
missed_tags=0;
noise_power=2.058; % Scalar value of noise power (sigma squared) in mv^2.

%The loop below (using the variable iruncount) spans most of the program
% and runs the simulation "sim_runs" times.
for iruncount=1:1

  %If, after the previous round, there are either collisions or packets
  % received in error, then another round of transmissions will be required
  % for all the packets involved in collisions or errors.
  % The loop below reduces the number of tags to only those in collision
  % or error and  provides for another round of transmission.
    for irounds=1:max
        collisionflag = zeros(1,no_oftags);   %collisionflag is a 1xno_oftags
        % array that will be used to indicate whether or not a particular
        % tag's packet was involved in a collision
        errorflag = zeros(1,no_oftags);       %errorflag is a 1xno_oftags
        % array that will be used to indicate whether or not a particular
        % tag's packet was received in error.

 % Randomly choose a slot for each tag
        slotchoice = zeros(1,1000);
        for tag =1:no_oftags
            slotchoice(tag)=randi(no_ofslots); %VECTOR 1Xno_oftags
        end

% This loop determines the number of messages that collide because their
% tags chose the same slot
       for j=1:no_oftags
            for k=j+1:no_oftags
                if slotchoice(j)==slotchoice(k)
                    collisionflag(j)=1;
                    collisionflag(k)=1;
                end
            end
        end
        collisions=collisions+sum(collisionflag);    % "collisions" contains
```

```
        % total number of collisions in all the simulation runs.  It's not
        % a very useful number except for diagnostics

% The large loop below determines if an uncollided tag's message is
% successfully transmitted or if noise causes a packet error

    for j=1:no_oftags
            if collisionflag(j)==1
    continue % If the tab was involved in a collision there is no
    % reason to check noise
    else
    tau = sigma*sqrt(-2*log(1-rand(1)));      %add Rayleigh fading
    noise_db = 10*log10(noise_power);
                for k=1:no_ofbits
    if errorflag(j)==1%skip loop if error has already been detected in tag

       break
    end
                    bitval=randi(0:1);
                    x_noise = wgn(1,1,noise_db);
                    if bitval==0 % This IF statement checks to see if a
                   % 0 was transmitted and the noise was large and positive

                      if x_noise>=5*tau
                         errorflag(j)=1;
                      end
                     else  %This ELSE statement checks to see if a 1 was
                         % transmitted and the noise was large and negative
                      if x_noise<-5*tau
                        errorflag(j)=1;
                      end
                   end
                end
            end
     end
      no_oftags=sum(collisionflag)+sum(errorflag);   %Calculate the number
      % of tags that will need to be transmitted for the next round

    total_rounds=total_rounds+1;
        if no_oftags==0
            break
        end
        if irounds==max
            overflow=overflow+1;
            missed_tags=missed_tags+no_oftags;
            break
        end
    end
message_errors=message_errors+sum(errorflag); % "message_errors" contains
% total number of message errors due to noise

end

total_rounds;%The average number of rounds per simulation will be
% total_rounds/sim_runs
```

```
overflow;    %This is the number of times per sim_runs that there was at
% least one tag's data still in error after the maximum number of rounds

missed_tags; %It's possible that more than one tag was still in error
% after the maximum number of rounds.  missed_tags is the total number of
% tags that were not correctly received after sim_runs simulations.

%Calculation of the percentage of error in the process
Perc_error = (100*missed_tags)/no_oftagsnew;
no_ofslots;

perc = Perc_error;
miss = missed_tags;

success_tags=no_oftagsnew-missed_tags;
successful = success_tags;

sl = no_ofslots;
```

REFERENCES

[1] Finkenzeller, K. (2010). "RFID Handbook. Fundamentals and applications in Contactless Smart Cards, Radio Frequency Identification and Near Field Communication". Wiley. https://repo.zenk-security.com/Magazine%20E-book/RFID_handbook.pdf.

[2] Murata. "Basic Knowledge of FRID". MuRata Innovator of Electronics. https://solution.murata.com/en-global/service/rfid-solution/basic/

[3] Lee, S. Joo, S. Lee, C. (2005). "An Enhanced Dynamic Framed Slotted ALOHA Algorithm for RFID Tag Identification https://ieeexplore.ieee.org/document/1540997

[4] Dheeraj, K. Kwan-Wu, C. Raad, R. (2010). "A Survey and Tutorial of RFID Anti-Collision Protocols". IEEE Communication Surveys & Tutorials. https://ieeexplore.ieee.org/abstract/document/5455790

[5] Zuliang,W. Shiqi, H. Linyan, F. Ting, Z. Libin, W. Yufan, W. (2018). "Adaptive and dynamic RFID tag anti-collision based on secant iteration". PLOS ONE. https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0206741

[6] Atlas RFID Store. (2016). "UHF RFID Tag Communications: Protocols and Standards". https://www.atlasrfidstore.com/rfid-insider/uhf-rfid-tag-communications-protocols-standards

[7] Cmiljanic, N. Landaluce, H. Perallos, A. (2018). "A Comparison of RFID Anti-Collision Protocols for Tag Identification". MDPI. https://www.mdpi.com/2076-3417/8/8/1282/htm

[8] Keni A. (2019). "Using CDMA/AIC to increase energy efficiency and reduce multipath effects in passive RFID Tag systems". Texas State University

[9] Doany, R. Lovejoy, C. Jones, K. Stern, H. (2016). "A CDMA-based RFID Inventory System. A CDMA Approach as a Solution for Decreased Power Consumption". IEEE International Conference on RFID. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7488023&tag

[10] Stern, H. Mahmoud, S. Stern, L. (2004). "Communication Systems". Pearson. First Edition

[11] Rappaport, T. (2015). "Wireless Communications. Principles and Practice". Prentice Hall. Second edition.

[12] Sanasi, P. (2021). “Extension of the Code Division Multiple Access/Adaptive Interference Cancellation protocol through analysis and simulation”. Texas State University. https://digital.library.txstate.edu/handle/10877/13504

[13] Vahedi, E. Ward, R. Blake, I. (2014). “Performance Analysis of RFID Protocols: CDMA Versus the Standard EPC Gen-2”. IEEE. https://ieeexplore.ieee.org/abstract/document/6716098

[14] Maina, J. Mickle, M. Lovell, M. Schaefer, L. (2007). “Application of CDMA for anti-collision and increased read efficiency of multiple RFID tags”. ELSEVIER. https://www.sciencedirect.com/science/article/abs/pii/S0278612508000113?via%3Dihub

[15] Wuu, L. Chen, Y. Hung, C. Kuo, W. (2009). “Zero-Collision RFID Tags Identification based on CDMA”. IEEE. https://ieeexplore.ieee.org/document/5283877

[16] Wikipedia contributors. (2022, January 19). Hash chain. Wikipedia. https://en.wikipedia.org/wiki/Hash_chain

[17] Bae, J. Song, I. Won, D. (2002). “A selective and adaptive interference cancellation scheme for code division multiple access systems”. ELSEVIER. https://www-webofscience-com.libproxy.txstate.edu/wos/ccc/full-record/CCC:000181042600003