Jammer Localization in Wireless Sensor Networks (WSN)

Panayiotis Vasiliou

University of Cyprus

Department of Computer Science

June 2023

University of Cyprus

Department of Computer Science

Jammer Localization in Wireless Sensor Networks (WSN)

Panayiotis Vasiliou

Research Supervisor

Dr. Vasos Vassiliou

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science at the University of Cyprus

June 2023

Acknowledgements

First and foremost, I would like to thank my thesis supervisor and mentor, professor Dr. Vasos Vasiliou for his invaluable assistance and guidance throughout this academic year. I would also like to express my gratitude towards PhD candidate Michael Savva for his assistance with all the necessary data and information he provided me with and wish him the best for his future. Lastly, I want to thank my beloved family and friends for supporting me and being by my side all these years.

Summary

The Internet of Things (IoT) has become an integral part of our daily lives in various institutions and domains for the past decades. Wireless Sensor Networks (WSNs) which form the backbone of the IoT are a fundamental component and they provide the essential data and relevant information that powers IoT applications. Since it is in the nature of man to cause damage for expediency and due to the importance, ubiquity, and interconnectedness of WSNs, the latter have been getting increasingly vulnerable to attacks. Among these attacks, jamming attacks pose a significant threat. In a jamming attack, a malicious actor disrupts communication within the network and tries to render it unavailable by flooding it with interference signals. Since the consequences of such an attack can be severe, actions need to be taken to mitigate as much damage as possible.

This paper focuses on one very important part of counter measuring the attack—jammer localization. Jammer localization is the process of determining or estimating the potential coordinates of the jammer within the network. This thesis presents a comprehensive review and comparison of five jammer localization algorithms, cross examining their performance across different types of network topologies, jammer types and jammer position scenarios to identify the best approach and reach some conclusions regarding the effectiveness of the algorithms.

It is also important to note that the performance of the algorithms depends on the quality of the information they receive, a point which is discussed later in the paper. The findings and insights from this work can be useful and contribute to the ongoing battle against jamming attacks by aiding in the development of effective countermeasures, thereby strengthening the overall security of WSN and IoT deployment.

Table of Content

Chapter 1

Introduction

1.1 Motivation

When the internet first emerged, human to human communication was the primary form of interaction for internet-connected devices. Most of the machines that had access to the internet still had to be operated by a human. For the past few decades however, this has not been the case, as we witness more and more devices interconnected with one another, exchanging information bringing us closer to the role of the spectator in the IoT era [1]. As a result, Wireless Sensor Networks (WSNs), which collect and interface the information between the sensors/ actuator in the IoT, have been getting more and more common [2].

Critical infrastructure sectors, such as chemical, energy, communication and even the healthcare and public health, are now relying on wireless sensor as a reality of our daily lives. Such crucial aspects of our daily lives can be disrupted by malicious jamming attacks which can cripple a part of the network or even render the whole network unable to transmit data. The consequences range from security compromises, hindered communication to complete system failure. This growing threat has compelled us in developing strategies that prevent the attacks or at least mitigate the damage as much as possible.

According to recent studies, it is estimated that the number of connected IoT devices could reach up to 45 billion and the numbers of attacks on wireless sensor networks have

skyrocketed. Only in in the first half of 2019, we observed an increase of more than 300% in the number of jamming attacks [4]. Given that cybercrime against companies amounts to a loss of nearly USD 6 trillion per year, and it could cost the world USD 10.5 trillion annually by 2025 there is a lot of money on the line and big companies are willing to invest heavily on anti-jamming measures [5].

Since WSNs are so ubiquitous and since they are comprised of sensors that are of low computational power and energy capabilities, they often become easy targets for adversaries that attempt to jam the network. To ensure the functionality, reliability, and integrity of the WSNs we have developed algorithms that try to estimate the location of the jammer within the network, the so-called jammer localization algorithms.
Our motivation arises to seek the best way we can leverage information from a network suffering a jamming attack, in an attempt to mitigate the effects of the jamming attack by estimating the geological x,y coordinates of the jammer within the network, so that further actions can be taken to restore normality to the network. We provide this information to the algorithms, and we obtain the estimated position of the jammer.

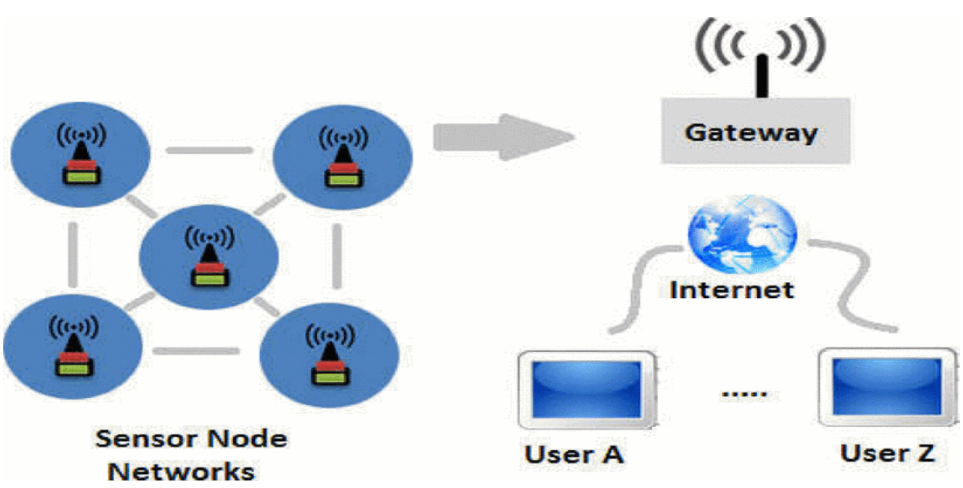


Figure 1.1 – Wireless sensor network (WSN) [3]

1.2 Goal

The main objective of this research is to investigate the effectiveness of various jammer localization algorithms, across different combinations of topologies, jammer types and jammer positions within the network. By cross examining five state-of-the-art algorithms, this study aims to draw meaningful conclusions regarding the accuracy of each algorithm.

By comparing the results we obtain from extensive tests and simulations, we can pinpoint the strengths and weaknesses of these algorithms, to determine the most reliable approach to localizing the jammer.

To further clarify the goal of this paper, we can break down the objective into a couple of useful research questions/ hypotheses.

- With 5 algorithms at our disposal, each one with its own pros and cons, how do they differ in terms of performance? Is mean accuracy sufficient for gaining a general understanding of their performance?
- How is each algorithm influenced from the conditions of the simulation and the quality of the provided data?
- Are these algorithms performing as expected? Is their performance solely based on how good they are or are they constrained by the information we have at hand?

By gaining a comprehensive understanding of the strengths and weaknesses of different jammer localization algorithms, this research will contribute by providing valuable insights and conclusions in implementing defence systems that select the best option available when it comes to jammer localization algorithms.

Ultimately, the outcomes of this research can contribute to the ongoing battle against malicious jamming attacks, which will get more common, more severe, and more complex as the world of IoT becomes increasingly integrated into our daily lives.

1.3 Methodology

This work employs a comparative experimental research design, and this is done by comparing the results of five jammer localization algorithms. It is important to point out that for a complete and meaningful evaluation of the algorithms, we had to treat them with fairness. This means that all experiments conducted are controlled and run under standardized conditions. Four out of the five algorithms follow a relatively similar baseline approach, while the fifth algorithm, which was developed completely from scratch too, has a distinct approach. Nonetheless, all of them were tested using the same

datasets and had the exact same information available to them, even when alternative information available within the data seemed intuitively more suitable. This is to ensure that all the algorithms have the same information to their disposal as the work focuses on how the algorithms behave with data obtained from a specific logic, named fuzzy logic, which will be explained later.

Initially, we started by getting a thorough understanding of what data we are going to work with as well as reviewing various surveys on the general topic of jammer localization such as the work done by Yanqiang Sun et al. [6]. The data we had at our disposal was information obtained on the condition of the network based on a fuzzy logic which produces an index, noting whether a node in our network is jammed or not.

Next, we conducted a comprehensive literature review to gather information about the existing jammer localization algorithms. This involved analysing papers, reports, articles, and relevant sources, to get an idea of the existing approaches towards localizing a jammer after the attack has been detected.

Based on the information and the knowledge gathered from the steps mentioned previously, we conducted a second, more comprehensive and extensive literature review to select a subset of the algorithms we researched previously in a way that we could pair them with the data we had in our disposal. This was important because a lot of algorithms were not going to work with what we had in hand, due to the nature of how the algorithms process information to obtain results. We were able to extract key details about the methodology of each algorithm, as well as its implementation and assumptions. Four of the five algorithms were supplied with a baseline implementation, which we then adjusted and fine-tuned to make them compatible with our data. As for the fifth algorithm, it was developed from scratch following a variation of the original algorithm proposed in the paper.

The data that our algorithms used, was provided in the form of excel files. This information was obtained by running various scenarios in the ContikiOS simulator. However, it does not concern us on how the data was obtained as this is not the purpose of the research.

Subsequently, we executed our algorithms on this data for all the possible scenarios. This covers different jammer types, different topologies, and different jammer locations. The

data consists of two topologies, four different types of jammers and sixteen different jammer locations. This means we run a total of 2x4x16x5 python scripts to cover all the possible scenarios. The scripts were performed in Visual Studio Code editor.

Regarding the data analysis techniques, we utilized the statistical metric of Euclidean distance between the algorithm's estimation and the actual jammer location. This allowed us to evaluate the performance of each algorithm based on localization error. By considering all possible combinations of environment parameters, we examined how each algorithm behaves with regards to different parameters.

It is also worth mentioning that this work was limited to topologies where the sink was in the middle. Additionally, the performance of all the algorithms solely relies on the quality of the data we tested them with, and more specifically with the fuzzy index of each node in the network.

1.4 Structure

In Chapter 1 we have provided the overall goal of this paper as well as giving the motivation behind the research. Moreover, we present the process of thought and the path we followed towards completing this paper.

In Chapter 2 we give a brief but adequate explanation on the part of detecting the jammer attack within the network, as this was the way the data for the algorithms was obtained. Additionally, we describe the structure of the data we were supplied with.

Chapter 3 offers a comprehensive review of the five state-of-the-art algorithms that were selected for our experiments. We explain how each algorithms works and how it processes the information from the datasets. Furthermore, we provide insight into how we decided which values would the parameters of the fifth algorithm get.

Next in Chapter 4, we give a quick reference to the four types of jammers that were used in the attacks from which we gained the data from. It is also worth noting that each jammer type has provided different results in the jamming detection phase, which has resulted in different results across our experiments.

In Chapter 5, we showcase the experiments we run across the various available combinations from topologies, jammers, and jammer positions. We give a brief explanation for each scenario, so that the reader can have a clear picture of what was happening behind the scenes while we obtained the results. Graphs for each scenario are presented and analysed in a meaningful way, facilitating the derivation of conclusions.

Finally, Chapter 6 proposes ideas and suggestions on future work that would greatly enhance the performance of the algorithms in similar scenarios.

Chapter 2

Detection Mechanisms and Simulation Data

The process of detecting a jamming attack is the initial step in taking effective measures against such attacks. A detection mechanism that actively monitors the network, to spot anomalies and unusual activity is what has provided the relevant information that will be used later in the process of localizing the jammer. Based on a set of parameters that determine the input, it decides which nodes are likely to be jammed. This chapter provides a high-level overview of the role of fuzzy logic in determining the jamming status of the nodes within the network and how that is connected to our research.

2.1 Detection Mechanisms

2.1.1 Fuzzy logic for Jamming Detection

Fuzzy logic is a framework that enables the representation and manipulation of imprecise values or concepts. In contrast to a more traditional mathematical framework, fuzzy logic gives us the opportunity to work with different degrees of truth, instead of relying on binary TRUE/FALSE values. By incorporating imprecise, uncertain, and incomplete information, fuzzy logic provides a bridge between natural and machine intelligence [7].

In general, fuzzy logic as a detection framework can make real-time decisions even with lacking information, therefore it is considered a suitable option for addressing this problem [8]. Since it can accept a combination of metrics, it can also represent the environment through if-then rules accurately [9]. Based on a set of classification rules, we can create fuzzy sets which resemble membership functions for the possible values that the variables can have.

In the work by Savva, M et al. [10], an intrusion detection technique which uses fuzzy logic as its detection mechanism is proposed. In their work, they employed a combination of the following metrics to generate the fuzzy index:

- ETX: Expected Transmission count (ETX) is the expected number of transmissions required to successfully transmit and acknowledge a packet.
- Retransmissions: The total number that a packet had to be transmitted over and over until it was successfully received.
- PDPT: Packets Dropped Per Terminal (PDPT) is the ration of the number of received packets that have not passed the Cyclic Redundancy Check, to the total number of packets received by the node.
- PDR: Packet delivery Ratio (PDR) is the ratio of packets that are successfully delivered to the destination to the total number of packets sent out by the sender.

Universe of discourse (uod)	Set	a	b	c	d
ETX	LOW	-4	-2	2	3
	MEDIUM	2	3	10	11
	HIGH	10	11	18	22
Retransmissions	LOW	-150	-100	500	600
	MEDIUM	500	600	1100	1200
	HIGH	1100	1200	4100	4500
PDPT	LOW	-10	-8	8	10
	MEDIUM	8	10	78	80
	HIGH	78	80	148	150
PDR	LOW	-0.5	0	0.45	0.5
	MEDIUM	0.45	0.5	0.85	0.9
	HIGH	0.85	0.9	1	1.05
Jamming Indicator (JI)	NO ATTACK	-0.05	0	0.25	0.3
	LOW	0.25	0.3	0.5	0.55
	MEDIUM	0.5	0.55	0.75	0.8
	HIGH	0.75	0.8	1	1.05

Figure 2.1.1 Values of Variables Used In the Definition of Membership Functions [10]

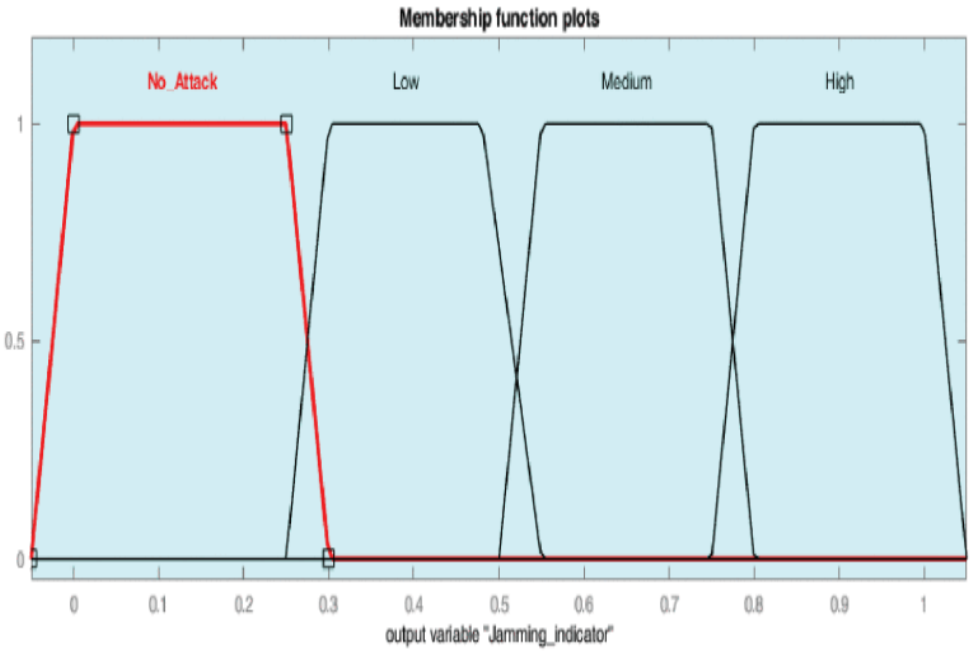


Figure 2.1.2 The trapezoidal Membership function plots for the output, Jamming index (JI). [10]

2.2 Simulation Data

2.2.1 Simulation Setup and Dataset Generation

The datasets used in this research were derived from simulations performed in the Contiki OS with the help of the Cooja simulator tool. By simulating a network topology as a grid area with 25 nodes with a central node acting as the sink, different scenarios were performed. In each scenario, there was a different combination of topology type, jammer type and jammer position with 2 different types of topologies (uniform and random), four different jammer types and 16 different jammer positions.

By simulating all these different scenarios, enough data was generated, which would reflect the conditions within the network producing a fuzzy index that indicates the likelihood of a node being jammed.

While it is important to highlight the importance of the fuzzy index in determining the likelihood of a node being jammed, we chose not to go into excessive and specific technical detail on this matter, as this is not the primary focus of this research. Fuzzy logic only serves as the technique that enables the bridge between the attack detection and the jammer localization in a wireless sensor network suffering an attack. With this concise explanation, we showcase how the necessary input information for the algorithms was generated, ensuring an extensive and comprehensive comparison and evaluation of their results.

In the following two screenshots, Figure 2.2.1 – Uniformly Distributed Topology and Figure 2.2.2 – Randomly Distributed Topology, we can see a graphical representation of how the two types of topologies, uniform and random respectively were set from the simulations. It is important to note that both topology types feature the same 16 jammer locations.

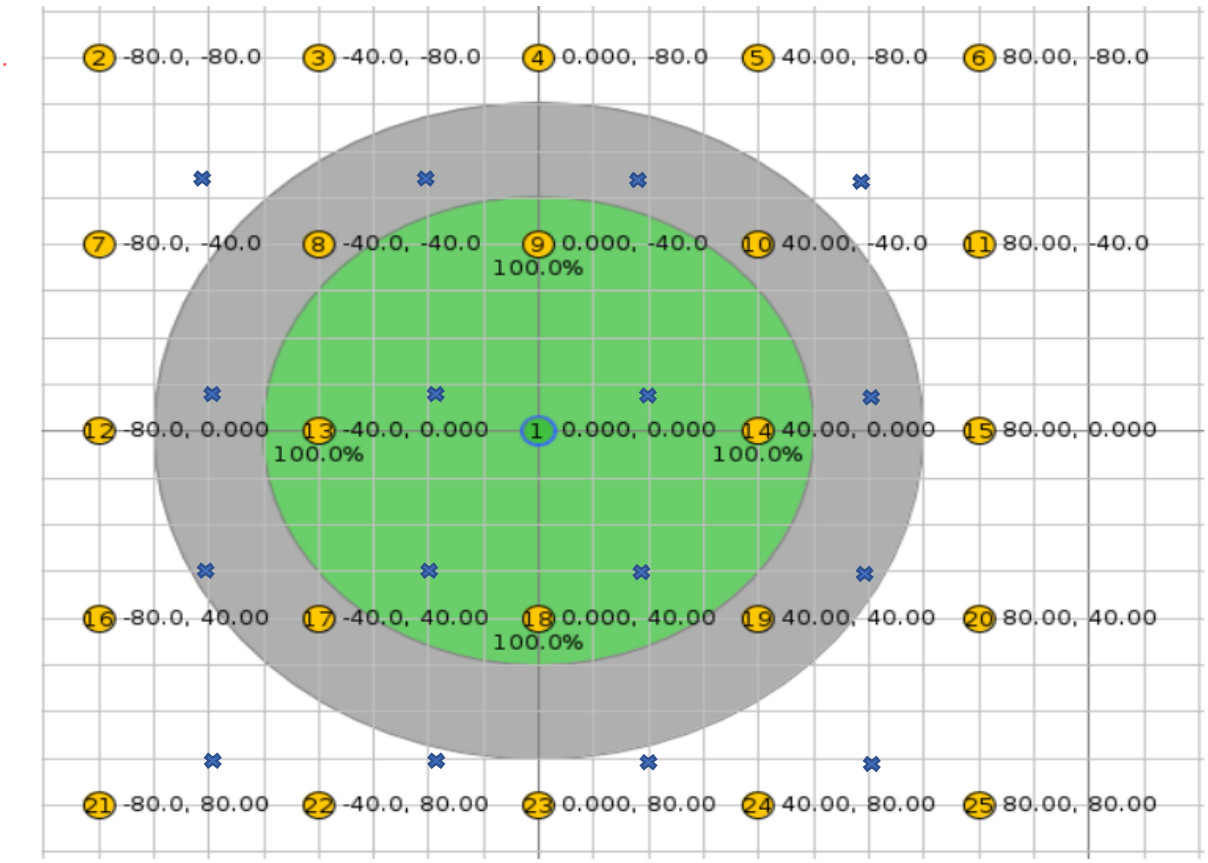


Figure 2.2.1- Uniformly Distributed Topology [10]

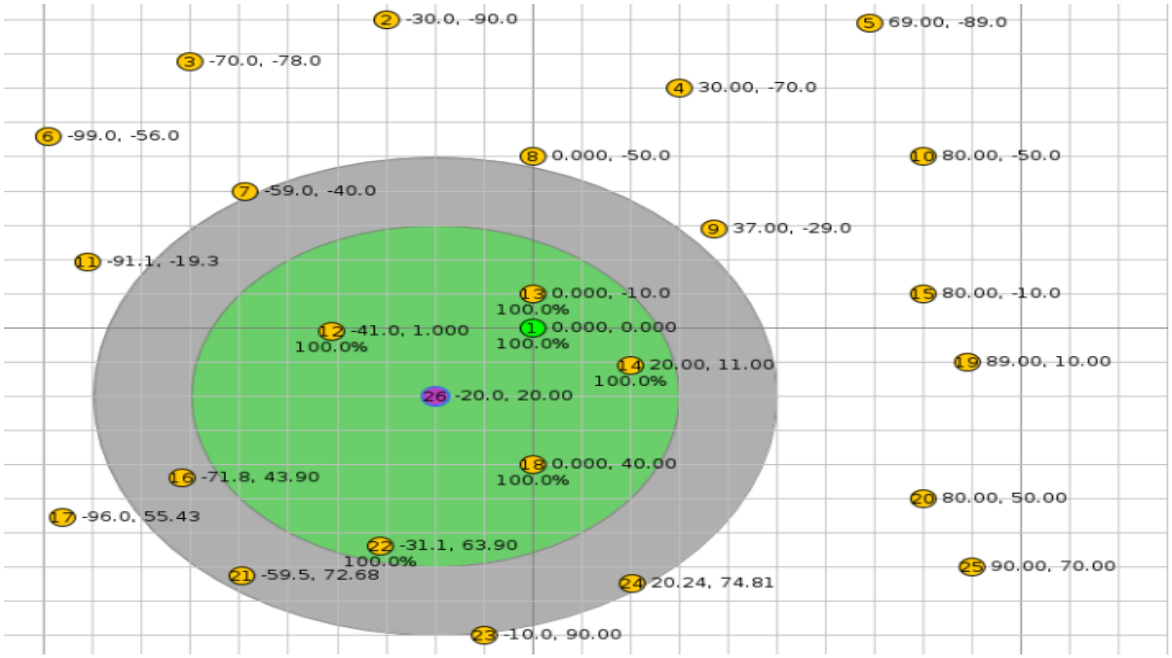


Figure 2.2.2 – Random Distribution Topology

In the following Table 2.2.3 – Simulation Data, we can see the form, structure, and length of the data in the way it was provided to us for this research with the full table used in one of the scenarios. Each scenario simulates one minute of traffic monitoring within the network. With 336 records, we have fourteen batches of nodes, with each batch holding 24 nodes. Every 24 nodes, the process starts over until a full minute has been captured.

				ETX	Retransmi	Fuzzy	
6.038416	20.54545	2	0	0.327954	0.041494	0.1354	0
4.298618	68.18182	3	1	0.207939	0.222683	0.1354	0
4.072945	101.9091	4	1	0.192372	0.350968	0.4	0
6.958457	40.81818	5	0	0.391421	0.118603	0.1354	0
3.309529	21.72727	6	0	0.13971	0.045989	0.1354	0
6.814834	51.72727	7	0	0.381513	0.160097	0.1354	0
3.980012	97.72727	8	1	0.185961	0.335062	0.3185	0
3.740753	127	9	1	0.169456	0.446404	0.4	0
6.921223	70	10	0	0.388852	0.229599	0.1354	0
2.160142	52.63636	11	0	0.060423	0.163555	0.1354	0
5.067873	51.72727	12	0	0.261004	0.160097	0.1379	0

Table 2.2.3 – Simulation Data

2.2.2 Filtered Data for Algorithm Evaluation

In the following screenshot Figure 2.2.4 – Snapshot of Data, we can see the relevant information that was used in our algorithms. While other approaches could make use of more information from the table, we chose to follow a more simplified approach, to treat the algorithms with fairness, by utilizing the same minimal information for all algorithms. Due to the simplicity of some algorithms, that do not need excessive amounts of information to work, we preferred to use that minimal information for all of the algorithms. This would

allow us to evaluate the performance of each algorithm and compare them in the fairest way possible, reaching safe conclusions about the accuracy of the algorithms in question.

	A	B	C	D	E	F	G	H	I	J	K
1					ETX	Retransmi	Fuzzy				
2	6.038416	20.54545	2	0	0.327954	0.041494	0.1354	0	T	N	TN
3	4.298618	68.18182	3	1	0.207939	0.222683	0.1354	0	F	N	FN
4	4.072945	101.9091	4	1	0.192372	0.350968	0.4	0	F	N	FN
5	6.958457	40.81818	5	0	0.391421	0.118603	0.1354	0	T	N	TN
6	3.309529	21.72727	6	0	0.13971	0.045989	0.1354	0	T	N	TN
7	6.814834	51.72727	7	0	0.381513	0.160097	0.1354	0	T	N	TN
8	3.980012	97.72727	8	1	0.185961	0.335062	0.3185	0	F	N	FN
9	3.740753	127	9	1	0.169456	0.446404	0.4	0	F	N	FN
10	6.921223	70	10	0	0.388852	0.229599	0.1354	0	T	N	TN
11	2.160142	52.63636	11	0	0.060423	0.163555	0.1354	0	T	N	TN
12	5.067873	51.72727	12	0	0.261004	0.160097	0.1379	0	T	N	TN
13	4.396573	58.09091	13	0	0.214696	0.184302	0.1354	0	T	N	TN
14	2.280332	97.63636	14	0	0.068713	0.334716	0.3167	0	T	N	TN
15	1.646334	68.36364	15	0	0.024979	0.223375	0.1354	0	T	N	TN

Figure 2.2.4- Snapshot of Data

The screenshot corresponds to a specific scenario in which the parameters were:

- Topology: uniformly distributed nodes.
- Jammer Type: Deceptive jammer.
- Jammer Position: 2 (-20, -60)

This means that these are the outputs of the simulation run in the Contiki OS, for one minute of recording the traffic within the network. The type of jammer used in this simulation is deceptive, it was placed in the second position out of the 16 different positions of jammers (-20, -60) and the nodes in the network were distributed uniformly (See Figure 2.2.1 – Uniformly Distributed Topology).

Due to the excessive amount of information, we can filter out most of the columns in our excel files, and only focus on the necessary data that is needed for the algorithms.

Therefore, the only columns that concern us are the following:

- C: The ID for each node, this column loops in the range of values of 2-25. This is used to correlate the node and its position, which is held in another excel file.
- G: This column is holding the jamming value for the node, which was obtained as the fuzzy index from the simulation.
- H: Column H, holds the information on whether a node is considered jammed or not jammed via binary values, 1 and 0 respectively.

To cover all the different combinations of jammer type/ topology type/ jammer position, we had an excel file (with the exact same format as the one shown in figure 2.2.4 – Snapshot of data), for each position (16 total positions in each topology) of the jammer. In total, this accumulates to a total of 2(topologies)* 4(jammer types) *16(different jammer locations).

In each of these files, there are 336 records, which add up to 14 batches of 24 nodes. This means that each algorithm estimates the possible location of the jammer 14 times. The method we chose to calculate the error for each algorithm, was via calculating the average error of all these batches. This is the safest way of assuming the accuracy of each algorithm because this resembles a realistic network monitoring scenario, in which some batches can provide relatively good information, while other might provide misleading and wrongful data. We assessed that the approach of using the worst-case scenario would not provide adequate information, as it was often leading to our algorithms completely missing the position of the jammer. The same stands for the best-case scenario which respectively led to estimations with zero localization error.

2.2.3 Visualization of Node and Jammer Coordinates

In figure 2.2.5 – Node coordinates and figure 2.2.6- Jammer coordinates, we can see the x,y coordinates of the nodes and the jammers respectively. It is worth noting that Node with ID 1 is the sink node, therefore its coordinates are (0,0).

	A	B	C
1	Node	X	Y
2	1	0	0
3	2	-80	-80
4	3	-40	-80
5	4	0	-80
6	5	40	-80
7	6	80	-80
8	7	-80	-40
9	8	-40	-40
10	9	0	-40
11	10	40	-40
12	11	80	-40
13	12	-80	0
14	13	-40	0
15	14	40	0
16	15	80	0
17	16	-80	40
18	17	-40	40
19	18	0	40
20	19	40	40
21	20	80	40
22	21	-80	80
23	22	-40	80
24	23	0	80
25	24	40	80
26	25	80	80

	A	B	C
1	Node	X	Y
2	1	0	0
3	2	-30	-90
4	3	-70	-78
5	4	30	-70
6	5	69	-89
7	6	-99	-56
8	7	-59	-40
9	8	0	-50
10	9	37	-29
11	10	80	-50
12	11	-91.1	-19.3
13	12	-41	1
14	13	0	-10
15	14	20	11
16	15	80	-10
17	16	-71.8	43.9
18	17	-96	55.43
19	18	0	40
20	19	89	10
21	20	80	50
22	21	-59.5	72.68
23	22	-31.1	63.9
24	23	-10	90
25	24	20.24	74.81
26	25	90	70

Figure 2.2.5 - Node Coordinates

	A	B
1	-60	-60
2	-20	-60
3	20	-60
4	60	-60
5	-60	-20
6	-20	-20
7	20	-20
8	60	-20
9	-60	20
10	-20	20
11	20	20
12	60	20
13	-60	60
14	-20	60
15	20	60
16	60	60

Figure 2.2.6 - Jammer Coordinates

Figure 2.2.5 - Node Coordinates shows the nodes' coordinates for both topology type, with the uniformly distributed topology shown on the left and the randomly distributed topology shown on the right.

Figure 2.2.6 - Jammer Coordinates shows the coordinates of the locations of the jammers.

Overall, in this chapter we provide an overview of the data obtained from the simulations. This includes a concise explanation of the simulation setup, the generation of the datasets and the filtering of the relevant information, which was then used in our algorithms. We employed a methodology that allows a fair comparison and evaluation of the algorithms' performance.

Chapter 3

Algorithmic Analysis

In this chapter, by going over how each algorithm works, we allow the reader to get an adequate amount of information needed regarding the logic behind the algorithms. Five state-of-the-art algorithms have been chosen to be tested and evaluated. As far as the first four algorithms are concerned, we were supplied with a simplistic baseline implementation which had to be tweaked in a way that would allow the process of the data we already had. It is also worth noting that these algorithms were also reviewed in the literature to ensure they are suitable to process the information we had at hand. The fifth algorithm was chosen after an extensive literature review based on how suitable it was regarding the data. Moreover, the last algorithm is much more sophisticated compared to the other four, therefore it is explained in much more detail. It is also worth mentioning that for our experiments we preferred to use range-free localization algorithms. As the name suggests, range-free algorithms do not rely on precise distance measurements between nodes for localization purposes. Instead, they utilize other information, such as connectivity or relative signal strengths, to estimate the location of the target node or jammer by using various geometric methods [13]. No need for

additional and special software support is necessary, compared to range-based techniques which comes with the benefit of better level of accuracy.

3.1 Related Work on Jammer Localization Algorithms

3.1.1 Centroid Localization Algorithm

Centroid Localization (CL) Algorithm is arguably the simplest algorithm out of the five algorithms reviewed in this research. This algorithm can calculate the possible jammer location, by taking the coordinates of the jammed nodes and averaging them [11]. It requires only a minimum of computations and little communication expenses compared to other algorithms and it is only relying on the binary information on whether a node is jammed or not.

$$\left(\hat{X}_{Jammer}, \hat{Y}_{Jammer}\right) = \left(\frac{\sum_{k=1}^{N} X_k}{N}, \frac{\sum_{k=1}^{N} Y_k}{N}\right)$$

Figure 3.1.1.1 – Centroid Localization Algorithm Formula [11]

When it comes to our experiments, CL only requires the information on whether a node is jammed or not. Hence, in our variation, we select data only from columns C and H which give the node's ID and its jamming condition respectively. After 24 nodes have been processed, the program calculates the averages as mentioned above, and then the relevant variables are reset as the program moves to the next 24-node batch. Due to its simplicity, CL is performing relatively good in scenarios where there is high node density, and where nodes are uniformly spread across the network. Lastly, CL can be considered the general baseline for all jammer localization algorithms, and almost all papers examined in the literature, regarding various localization algorithms, were comparing their findings with CL.

3.1.2 Weighted Centroid Localization

Much like CL, Weighted Centroid Localization (WCL) is still using averages to estimate the jammers position, with the only difference being that it also uses weights to achieve better results [12]. While there are many different options to use as weights, in the WCL variation that we were supplied with, the fuzzy index of each node was used as a weight metric. While this isn't a bad option, it caps the localization capabilities of the algorithm since the performance is completely reliant on the fuzzy index. This means that a faulty fuzzy index, or a fuzzy index that is not particularly good, will lead to errors, compared to using the distance as the metric of weight or a combination of both. Moreover, to ensure fairness in the comparison of the algorithms, we used only the fuzzy index from nodes that were recorded as jammed. This means that WCL had the exact same nodes to work with, with the CL algorithm. As a threshold, jammed nodes are the nodes whose jamming index value is greater than 0.575. In our experiments, we used the same approach in how the data is processed with the one mentioned in 3.1.1. This also includes resetting the relevant variables so that the program can safely proceed to the next batch of nodes, as well as how the average error for the whole scenario is calculated.

$$P_j''(x,y) = \frac{\sum_{j=1}^{n}(w_{ij} \cdot B_j(x,y))}{\sum_{j=1}^{n} w_{ij}}$$

Figure 3.1.2.1- WCL formula [12]

3.1.3 Double Circle Localization Algorithm

Double Circle Localization (DCL) is an algorithm that is being used in jammer localization with the usage of two geometrical concepts: minimum bounding circle (MBC) and maximum inscribed circle (MIC) [14]. The baseline edition of DCL that we had, was using these two concepts in a slightly different way than the one described in the literature review. It works by creating two convex hulls, one for jammed nodes and one

for non-jammed nodes, whereas the original proposed idea is to create the MBC to contain only non-jammed nodes and have MIC as another circle within the convex hull that the jammed nodes form. It involves comparing the positions of the jammed and non-jammed nodes by analysing their respective convex hulls and minimum bounding circles. Our algorithm, after creating the two convex hulls mentioned above, generates an MBC for each hull. At the end we calculate the relative displacement between the two MBCs. The resulting vector (x, y) provides an indication of the jammer's position relative to the overall network. Once again, after each calculation, the program resets the relevant variables and proceeds to get the next batch of nodes. It is also very important to note that this implementation of DCL has a major flaw. Due to its nature of constructing convex hulls and circles, there are scenarios in which the program does not come with an estimation, as there are not enough points in the dataset to create a convex hull. There need to be at least three jammed and three not jammed nodes, so that each convex hull can be created. This means that in scenarios where there isn't much available information, this algorithm fails to provide an estimation in some of the batches, or completely fails to provide an estimation at all. This is also discussed in our conclusions.

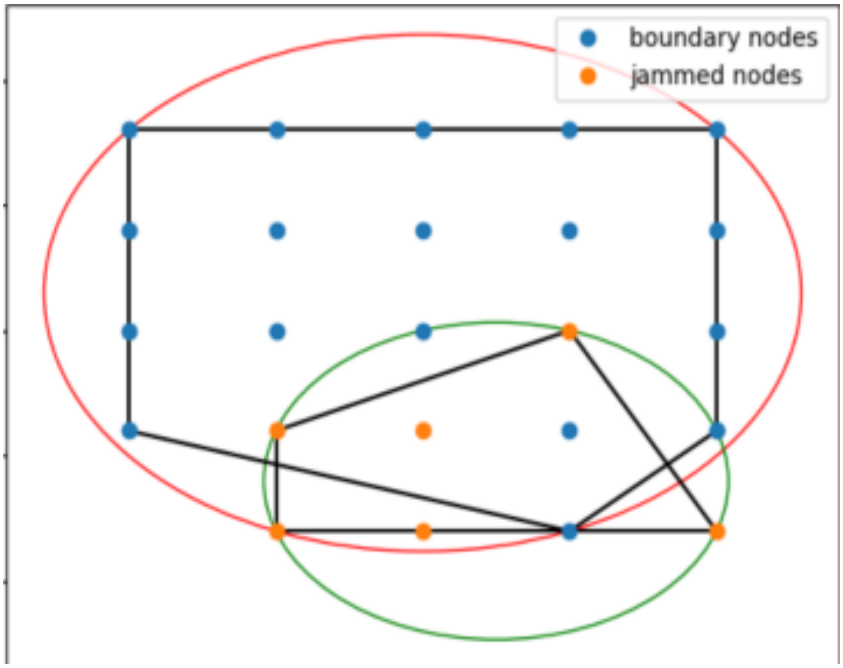


Figure 3.1.3.1- DCL Algorithm

3.1.4 Virtual Force Iterative Localization

Virtual Force Iterative Localization (VFIL) aims to estimate the possible location of the jammer by modelling the network as a system of interactive particles, which exercise push and pull forces towards one another. By utilizing network connectivity, in the work described by Hongbo Liu, et.al [15], the algorithm initially calculates the centroid using the CL algorithm (described in 3.1.1) and then it iteratively re-estimates the position until it's close to the real position, using a moving step. In the code we had to our disposal, after we initially calculate the location using CL, we re-estimate the location by examining the push and pull forces of the particles/nodes. The logic behind this, is that jammed nodes exert pull forces, while non-jammed nodes exert push force. By summing all the push forces and pull forces respectively, and by using a special formula, we can calculate the Joint force, which as a value helps in changing the initial calculation to a better, more accurate and precise one. Much like all the previous algorithms, we make sure to reset the relevant variables after each batch is processed before moving to the next one. In general, VFIL attempts to "fix" the estimation made by the CL, by incorporating extra calculations after the CL algorithm makes the estimation.

$$F^{i}_{pull} = [\frac{X_i - \hat{X}_0}{\sqrt{(X_i - \hat{X}_0)^2 + (Y_i - \hat{Y}_0)^2}}, \frac{Y_i - \hat{Y}_0}{\sqrt{(X_i - \hat{X}_0)^2 + (Y_i - \hat{Y}_0)^2}}],$$
$$F^{j}_{push} = [\frac{\hat{X}_0 - X_j}{\sqrt{(\hat{X}_0 - X_j)^2 + (\hat{Y}_0 - Y_j)^2}}, \frac{\hat{Y}_0 - Y_j}{\sqrt{(\hat{X}_0 - X_j)^2 + (\hat{Y}_0 - Y_j)^2}}]$$

Figure 3.1.4.1- Pull and Push Formula

$$\mathbf{F_{joint}} = \frac{\sum_{i \in J} \mathbf{F^{i}_{pull}} + \sum_{j \in B} \mathbf{F^{j}_{push}}}{\left| \sum_{i \in J} \mathbf{F^{i}_{pull}} + \sum_{j \in B} \mathbf{F^{j}_{push}} \right|}$$

Figure 3.1.4.2- Joint Force Formula

These range-free algorithms offer different approaches for estimating the location of jammers based on the network connectivity and geometric methods. While most of them are not implemented in the exact same way they are proposed in the literature, they are used as a good baseline example of how different approaches behave in scenarios from which the data is being collected by using fuzzy logic. In the following section, we will delve into the details of the Particle Swarm Optimization Algorithm, a more sophisticated approach that leverages swarm intelligence to tackle the jammer localization problem.

3.2 Particle Swarm Optimization Algorithm

3.2.1 Particle Swarm Optimization, A General Review

As a concept, Particle Swarm Optimization (PSO) is very similar to a genetic algorithm because it is initialized with a population of random possible solutions. The difference between PSO and genetic algorithms, is that each one of those candidate solutions is given a velocity and the ability to flow through the search space [16]. It is inspired by the social behaviour of fish schooling and bird flocks. In general, PSO is considered a metaheuristic optimization algorithm, and it aims to find the optimal solution in a problem by iteratively adjusting the position and velocity of candidate solutions, until a mutual convergence is reached. Based on individual experience of each particle, as well as a collective knowledge of the whole herd, acting as cognitive and social components respectively, the particles tend to move to the best solution, which resembles the jammer's location. To adapt PSO for the jammer localization problem, we modified the objective function, which basically uses a fitness function to define the quality of a particle, e.g., the solution.

Inspired by the work presented by Liang Pang et.al [17] we developed a variation of the algorithm, that would better suit the problem of localizing the jammer in wireless sensor networks with regards to our specific data and simulation experiments.

In the variation of the PSO algorithm that we developed, since it was developed from scratch, we swapped the stop criteria of convergence with a maximum number of iterations. Normally, PSO stops iterating as soon as the convergence reaches a value smaller than a stop criterion (usually a very small positive value [17]). After extensive experiments and trials, we noticed that for a fair number of scenarios, the convergence would never occur, since the stop criteria

was never reached. This is caused by the quality of the data obtained from the simulations described in 2.2. Therefore, we had to adapt to this problem, and we did that by setting the iterations as a parameter. This comes at a cost of extra execution time and resource allocation, but our algorithm manages to come with relatively good results and average execution time.

$$\boldsymbol{v}_i(t) = \boldsymbol{v}_i(t-1) + \varphi_1(\boldsymbol{p_l} - \boldsymbol{x_i}(\boldsymbol{t-1})) + \varphi_2(\boldsymbol{p_g} - \boldsymbol{x_i}(\boldsymbol{t-1}))$$
$$x_i(t) = x_i(t-1) + \boldsymbol{v}_i(t).$$

Figure 3.2.1.1 – Velocity and Position Equation [17]

In the PSO algorithm particles are moving through a socio-cognitive space, and therefore are affected both by their individual knowledge (cognitive component) and by the collective knowledge from all other particles (social component). The system is dynamic, meaning that these particles are constantly changing positions and velocities in an attempt to achieve the best fitness value, which is used to evaluate all solutions. The direction of movement is calculated with a formula that takes into consideration the current position and velocity, the best solution that the individual has found (cognitive component) and the best solution found in the entire herd (social component) [17].
As the equation in Figure 3.2.1.1 suggests, we use two effective parameters, φ1 for the cognitive component of the equation and φ2 for the social component of the equation. The symbols ρι and ρg resemble the local best and global best positions found so far respectively. Each particle is a candidate solution and to evaluate the quality of that solution we calculate its fitness value. More information follows on how the program is structured and how it works.

3.2.2 Implementation of PSO

Our program begins by creating and initializing particles in the search space, by assigning them a random position (x, y) and a random velocity (xVelocity, yVelocity). The program picks the batch and begins processing it. After picking a batch, the iterations begin and, in each iteration, we do the following tasks:
1) Pick the first particle from the list of particles.

2) For that particle, calculate the minimum covering circle, which encloses all the nodes that are jammed. This basically gives us the fitness value for the particle in question. The minimum covering circle, is defined by the furthest jammed node. After thorough thinking, we came up with the idea of using the distance from the particle's position to the furthest jammed node as the fitness value for the particle. Since PSO is a metaheuristic algorithm, we transform the problem of localizing the jammer to a minimization problem, in which the smallest fitness value is the best candidate solution, e.g., the position most likely to resemble the real location of the jammer. The Euclidean distance was used as they metric to calculate the distance between the furthest node.

3) Whenever we find a new, smaller minimal covering circle, this means we have found a better new solution, at least locally. This means that we check whether the new circle calculated is smaller than the previous smallest circle, and if so, we update the best local solution, by setting it to the one just discovered. This means that we keep track of the position and fitness value of the particle, to be able to access them in the comparisons. This is done to calculate the cognitive component.

4) When the circle for the particle is created, and the personal fitness value is calculated and after we update the personal best solution, if necessary, we need to check if we have a new global solution which is better than the previous global solution. We can think of this as having the best values for all the particles, with one of them being the global best value, and comparing the new personal best value with this global best value. This is done to calculate the social component.

5) As soon as these calculations are made, we can proceed to update the velocity and the position of the particle in question, by using the equation shown in Figure 3.2.1.1. The order in which we do these updates is very important, as the velocity needs to be updated first before updating the position, since the position is directly affected by the velocity.

6) After the velocity and position of the particle has been updated, we proceed to move to the next particle, and perform the steps 2-5 for the new particle in question.

7) As soon as all the particles have been processed, and all the possible solutions have been calculated, the program will move to the next iteration. This means that the particles start with the position and velocity that they had at the end of the last iteration. The steps 1-6 are repeated until all of the iterations have been completed.

8) As soon as the last iteration finishes, the program output the global best solution, which is the most accurate estimation of the jammer.

9) The program resets all the relevant variables, including the position and velocity of the particles, and then moves on to the next 24-node batch, in which the steps 1-8 are repeated, until all 14 batches have been processed.

This means that we have a total of 14(node batches) *24(nodes per batch) * 24 (iterations) * 60 (particles) iterations until the program completes.

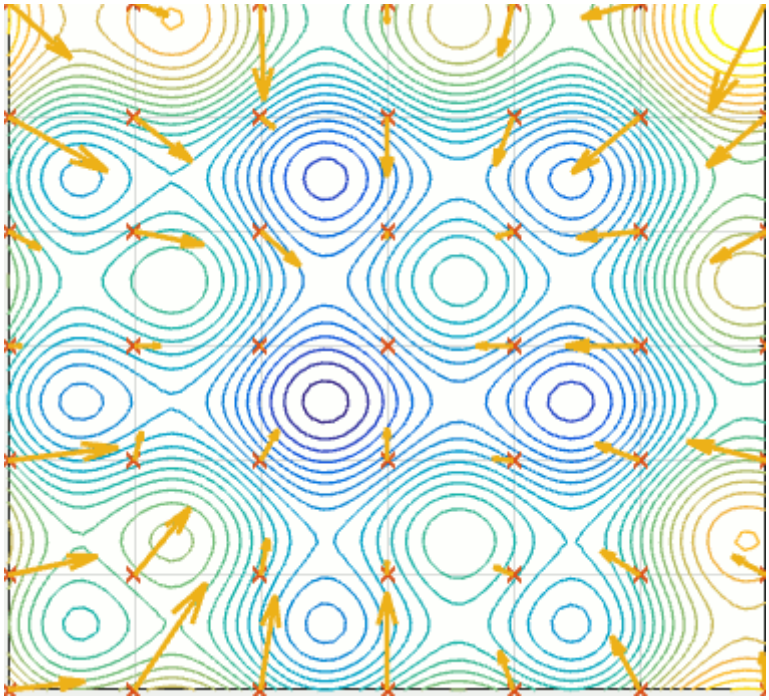


Figure 3.2.2.1 – Animation of particles

3.2.3 Parameter Selection for PSO

In this subchapter, we give some insight into the important aspect of parameter selection that took place in the PSO algorithm. This is very important, as the values of the parameters dramatically impact the overall performance of the algorithm. This includes the convergence speed, the execution speed, and the accuracy of the algorithm.

In our research, we focused on two primary parameters: the number of iterations and the number of particles. These two parameters play a significant role in how the algorithm performs. To determine the optimal values for these parameters we have conducted numerous investigations, by employing several experiments and performance evaluations. By

thoroughly analysing the impact of the number of particles and iteration we can decide which values give the best trade-off between convergence behaviour, solution quality and execution time.

Here are three graphs, that show the relationship between iteration number, particle number, average error, and CPU execution time. As you can notice, in each one of the three graphs, the number of iterations remains the same, increasing it from 5 to 10 and then to 20, 40 and 60. The particles in each graph slowly increase (X axis). This shows how average error is dropping as both particles and iterations are growing as well as how CPU time increases. The experiments stop when the execution time reaches a non-viable value, which is around the 4 second mark. As mentioned above, the average error is the average Euclidean distance between the algorithm's estimation and the real jammer location. CPU time is expressed in seconds.

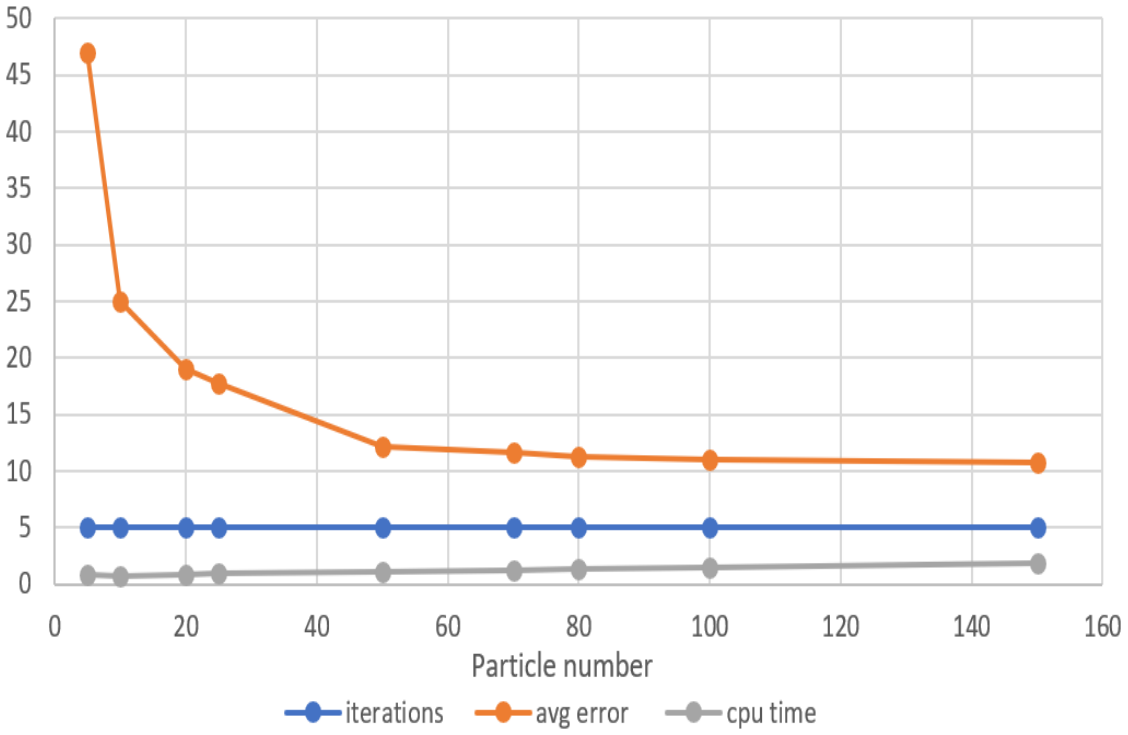


Figure 3.2.3.1

In figure 3.2.3.1, we keep the number of iterations steady at 5. By slowly increasing the number of particles, starting from 5 up to 150, we can see how the average error decreases and how CPU time increases. At around 50-80 particles, the average error starts to stabilize, meaning that using more particles than that gives no benefit to the average error and only

increases CPU execution time. Based on this, we start to get an idea on how the number of particles directly affects the average error and the CPU time.

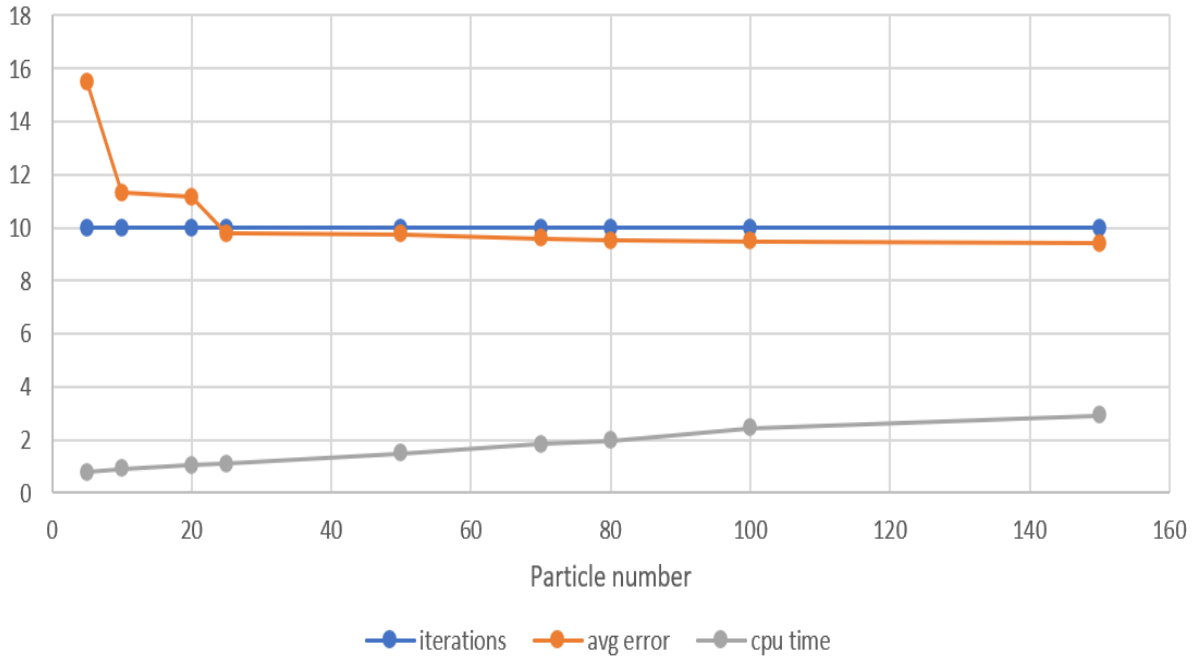


Figure 3.2.3.2

The same stands for figure 3.2.3.2 but this time, the number of iterations is increased to 10. This has a noticeable effect on the average error since it starts from much lower (15.8) compared to previously (47). Once again CPU time starts to increase drastically at around the 60-particle mark. Moreover, there is very little decrease in the average error as the number of particles goes above 60. We can conclude that the number of particles can only decrease the average error up to a certain point. By increasing the number of iterations, we manage two things: First, the starting errors from small number of particles is much lower compared to using the same number of particles but with lower iterations. Second, we manage to lower the minimum error much faster. In other words, in Figure 3.2.3.1 we can see that the error reaches 10 at almost the maximum number of particles (150 particles) whereas in Figure 3.2.3.2 we can see that the average error reaches 10 and even drops to smaller values much sooner, near 25 particles.

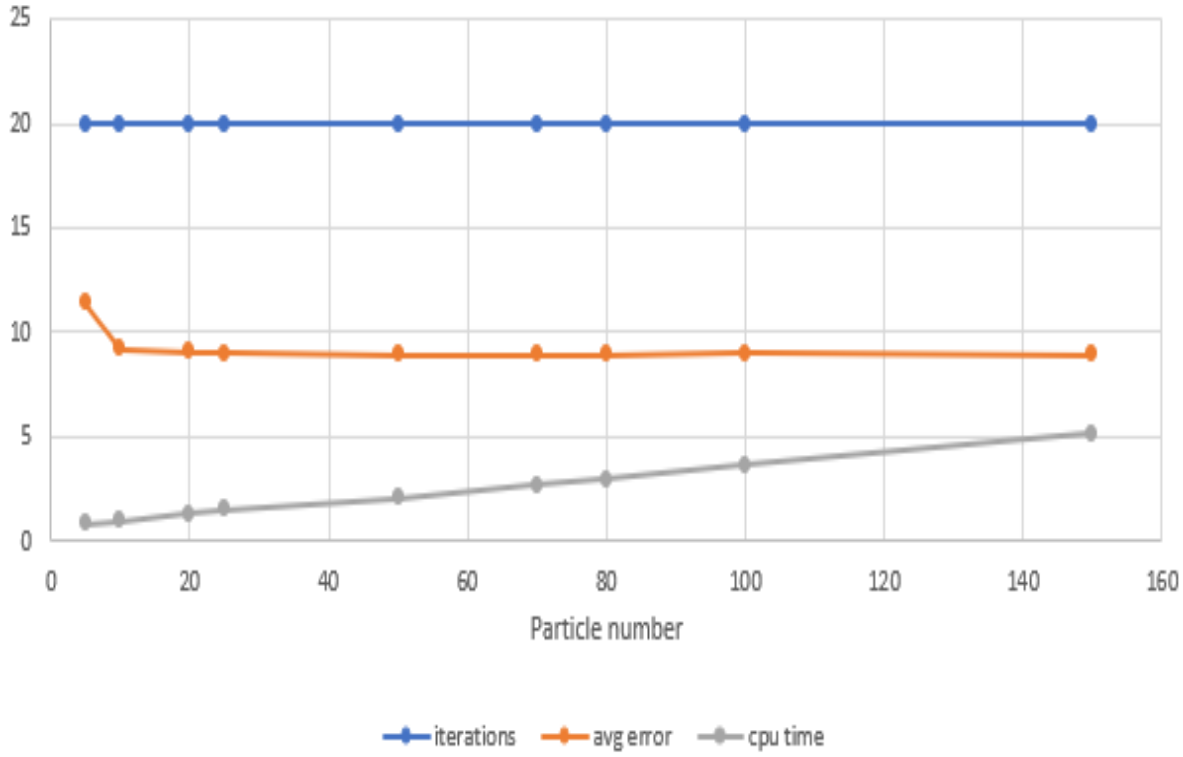


Figure 3.2.3.3

Similarly, Figure 3.2.3.3 shows how increasing the number of iterations drops the error, for the same number of particles. The error for 5 particles is again lower than both previous numbers of iterations (10 and 5). However, in this graph the problem with execution time becomes more obvious, as it skyrockets to 5 seconds at maximum number of particles (150). Yet, the trend that after around 55-60 particles the average errors stops dropping, still applies. At this point, we had a good idea on how the number of particles and iterations influence the average error and execution time. If we wanted to keep a relatively low and viable execution time, while still maintaining low errors, we had to use iterations in the range of 20-30 and particles in the range 40-70. Anything below or above these ranges, gives either big error or big execution times.

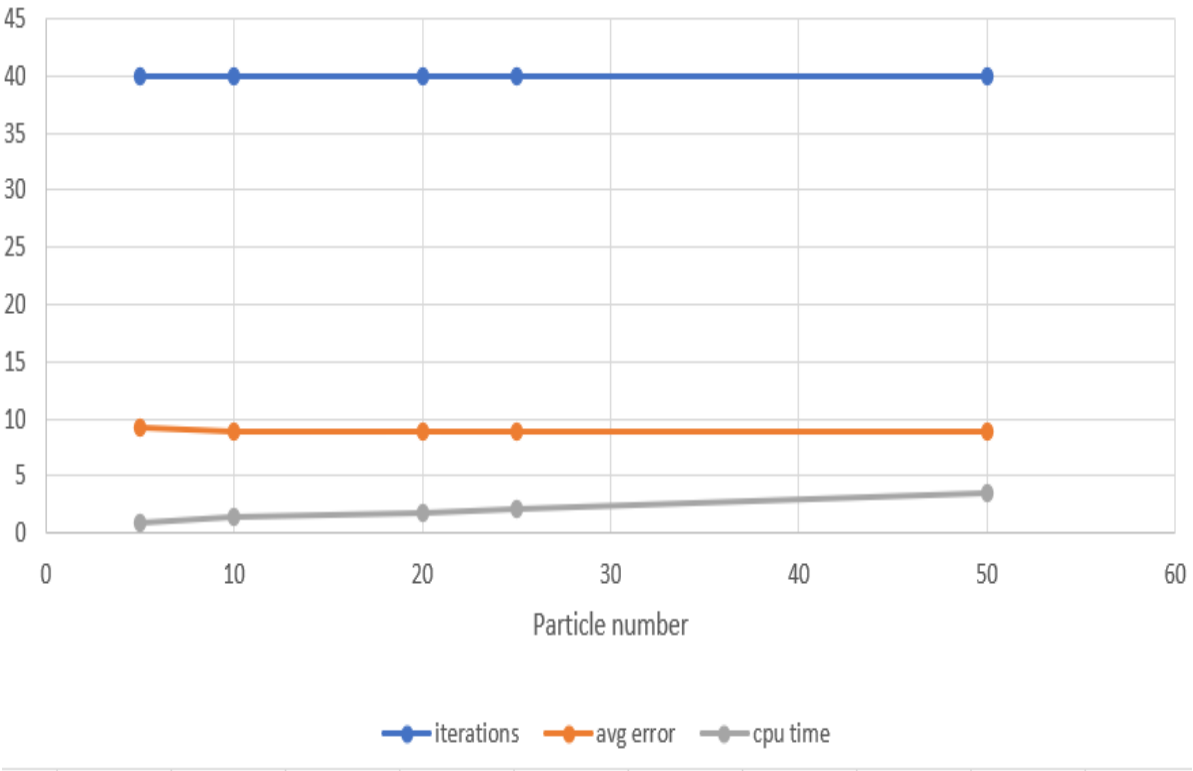


Figure 3.2.3.4

Figure 3.2.3.4 shows how average error and CPU time behave when the value of iterations is set to 40. We observe that the starting error for only 5 particles is very small at around 9.9. However, this value changes very little when more particles are added, while the CPU time increases dramatically, even when the extra particles added are only a few (notice how execution time reaches 5 seconds with only 50 particles). This led to the inclination that after a certain point, increasing the number of iterations does not provide a lower error, or at least an error that is low enough to justify the big execution number.

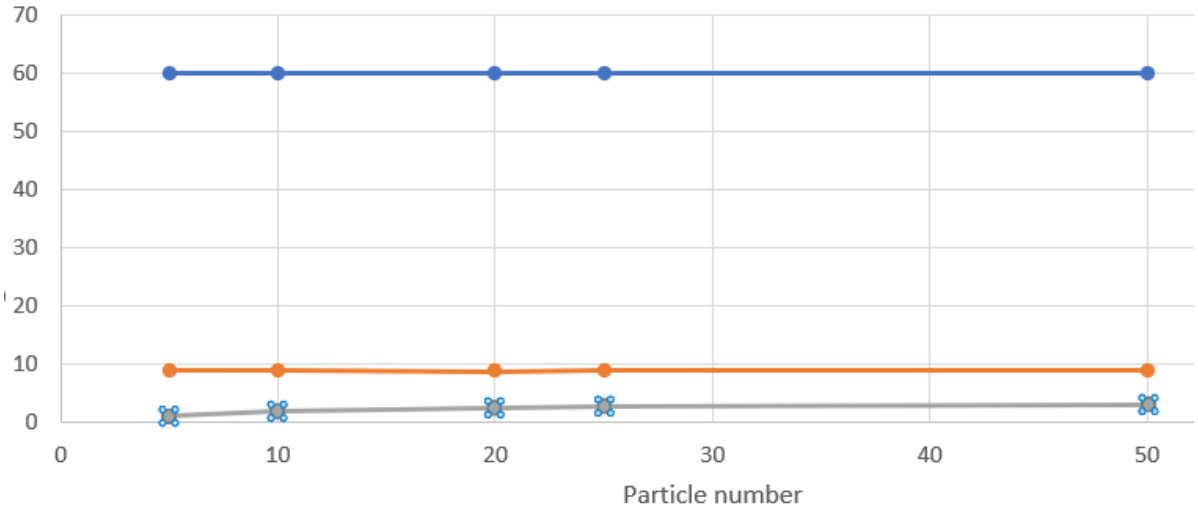


Figure 3.2.3.5

Finally, figure 3.2.3.5 showcases that there is absolutely no change in the average error when the number of iterations is set to 60. This means that our previous assumptions on the number of iterations were correct, and that we need to use a smaller number to maintain a good balance between execution time and average error, combined with the correct number of particles.

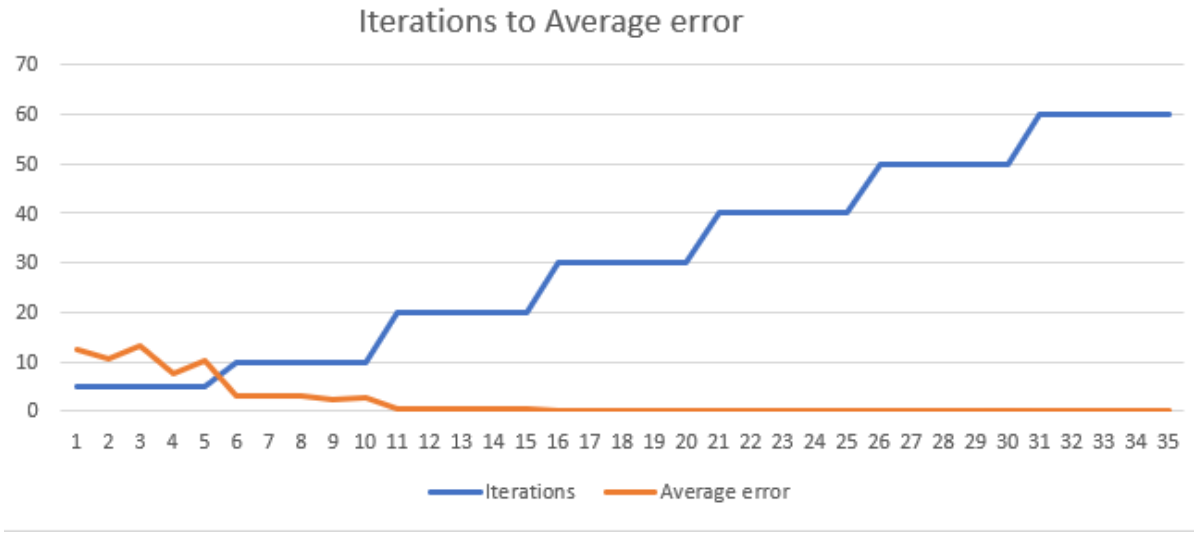


Figure 3.2.3.6

In Figure 3.2.3.6 we plot the number of iterations to the average error, which aids in getting a clearer understanding of the impact that the iterations have on the average error. For this graph, the number of particles was set to 30, which is a number in the middle of the range of particles (as discussed previously, the maximum number of particles is around 60). The graph

shows that as the number of iterations exceeds 25, there is practically no difference in the average error.

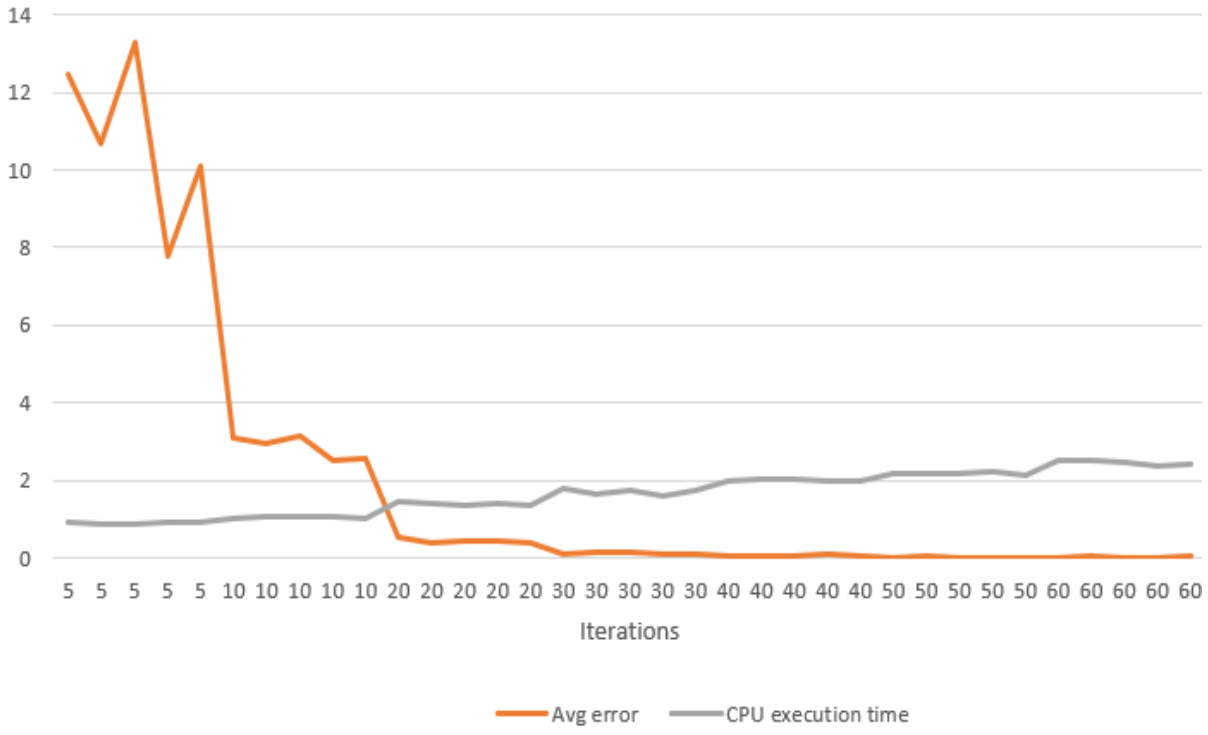


Figure 3.2.3.7

In figure 3.2.3.7, we demonstrate the effect of the number of iterations to the average error and CPU execution time, again in general picture. However, this time we simulate the experiment 5 times for the same number of iterations before incrementing it to the next value. This is to show that there are fluctuations due to the random nature of the algorithm. This is caused by the randomness in which particles are assigned an initial position and velocity and it is important to note that in some cases the algorithm was more fortunate regarding the placement of the particles, while in some other the placement was not that favourable. In any case, the experiment was tested multiple times so that the randomness does not give faulty results, or results that do not accurately represent the performance of the algorithm. We can clearly see that after the number of iterations exceeds the range of 20-30, there is no difference to the average error, and CPU execution time increases drastically.

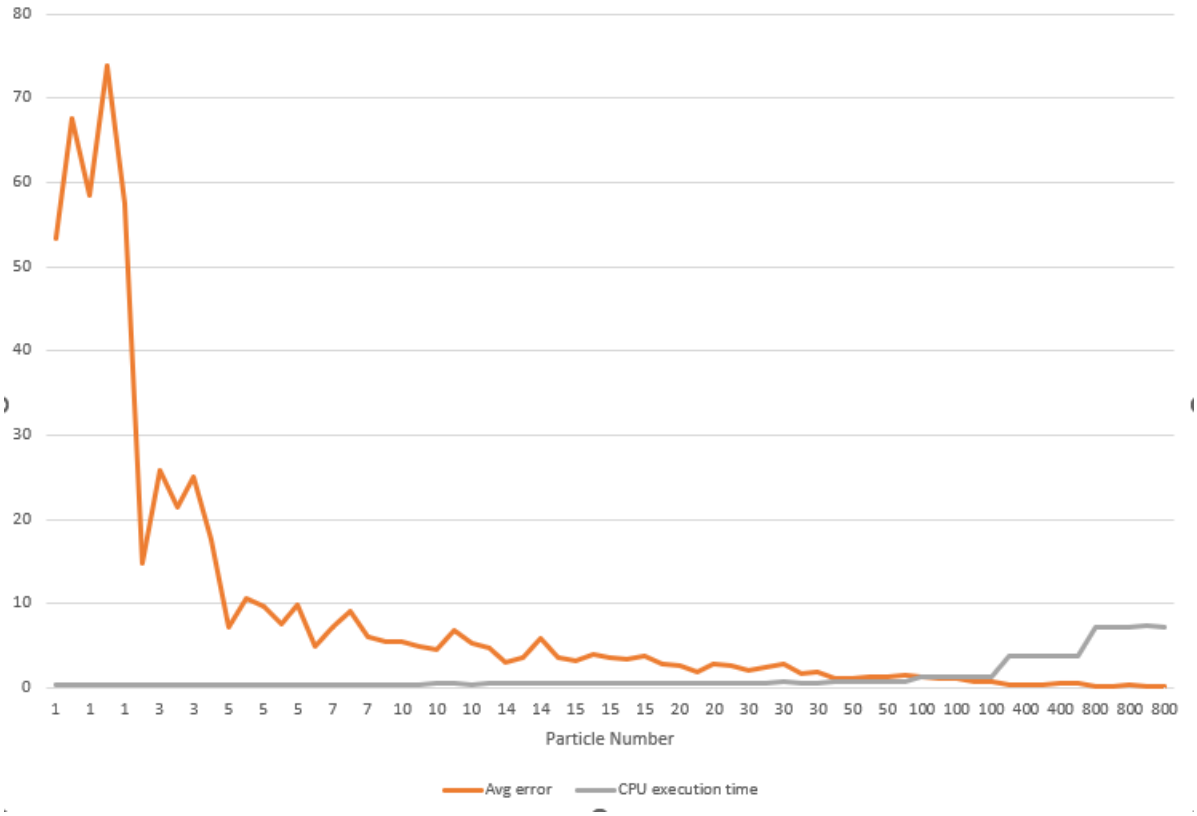


Figure 3.2.3.8

Lastly, in Figure 3.2.3.8 we show how different values for the number of particles affect the average error and CPU execution time of our algorithm. Similarly with the previous example, we keep the number of iterations steady at 10, which is a number that allows us to isolate the effect of particles on CPU execution time and average error, by producing results that represent it as close to reality as possible. Moreover, we use the same number of particles more than once, to once again show that the algorithm behaves differently in the same scenario, due to its randomness. It is obvious that after a certain number of particles, in the range of 50-80, there is very little difference in the decrease of the average error. Moreover, the trade-off between the average error and CPU execution time, after more than 80 particles are generated, does not justify in any case the increase in particle number.

In this chapter, we thoroughly investigated the importance of selecting the right value for two key parameters of the PSO algorithm, number of particles and number of iterations. By conducting a series of experiments, we aimed to reach a conclusion and decide which will be the values that we will be using to test our algorithm in all the scenarios of this research. By

considering the trade-off between CPU execution time and average Euclidean distance error, we demonstrate how the optimal value was decided for both parameters. The experiments demonstrated that the number of particles directly affects the average error and CPU time, with a diminishing return on error reduction beyond a certain particle count. Increasing the number of iterations, on the other hand, allowed for lower initial errors and faster convergence to a minimum error. However, there was a point of diminishing returns where additional iterations did not significantly improve the average error but resulted in increased execution time. Based on the results after the presented comprehensive investigation, we have concluded that to maintain a balance between execution time and average error, we recommend using a relatively low number of iterations in the range of 20-30, combined with an optimal number of particles ranging from 40 to 70. Values below or above these ranges result to suboptimal results. It is also worth mentioning that the random nature of the algorithm is producing variability in the performance of the algorithm, due to the initialization of the particles. To counter this, we conducted multiple experiments for the same value for each parameter, to produce results that correctly represent the true performance of the algorithm in general scenarios. In the following experiments, in which we test the algorithms on all the possible scenarios, we used 24 iterations and 60 particles.

Chapter 4

Jammer Types

4.1 Different Types of Jammers

Like previously mentioned, in our experiments we tested the performance of our algorithms across different topologies, jammer types and jammer locations. In this chapter, we briefly cover the different types of jammers that were used in the Contiki OS for generating the necessary data that we used in our experiments. This is a crucial detail, because different types of jammers force our algorithms to produce different localization errors. The results from different jammer types can provide valuable information regarding how good or bad each algorithm performs. These conclusions add up to the whole point of this research, which is to comprehensively review the performance of various range-free localization algorithms. The data we had in our disposal consisted of four different types of jammers. The work presented in [18] does an incredible job of describing the different jammer types that are being used. The first three types of jammers belong to the active jammers category, while the last type falls under the category of reactive jamming.

4.1.1 Constant Jammer

A constant jammer is a type of jammer that continuously emits a radio signal. A constant jammer does not follow any MAC-layer protocol and attempts to hold a channel busy by sending random data. It is seemingly the simplest jammer out of the four jammers reviewed in this chapter and it is the easiest to implement and the easiest to detect. Moreover, it requires a lot of energy to function because it keeps emitting signals non-stop.

4.1.2 Deceptive Jammer

Contrary to the Constant Jammer, a deceptive jammer is sending regular packets in between the normal flow of packets in the network channel. Due to this fact, this jammer

is much harder to infer while keeping a relatively easy implementation. However, a deceptive jammer is requiring a lot of energy to function properly.

4.1.3 Random Jammer

A random jammer instead of using a lot of energy to continuously emit signal data, can alternate between sleeping and jamming modes. This results in much lower energy consumption which is the major flaw of the two types of jammers mentioned right above. During its jamming phase, it can behave either as a constant jammer and as a deceptive jammer. It is worth noting that from our experiments, this type of jammer proved to be the most dangerous of all, causing the biggest combined average error. More information on this will be discussed in the conclusions chapter.

4.1.4 Reactive Jammer

In contrast with the previous three types of jammers mentioned right above, a reactive jammer follows a different approach. Since this type of jammer belongs to the reactive family, it does not jam the network when there is no traffic observed. This makes it consume less energy and at the same time increases the difficulty of detection. However, it is challenging to design and works on a single channel.

Chapter 5

Comparative Analysis of Range-Free Jammer Localization Algorithms

Chapter 5 features a comprehensive comparative review of the five state-of-the-art jammer localization algorithms presented and explained in chapter 3. The essence of the study is to evaluate the performance and effectiveness of these algorithms, by examining various datasets which are explained in chapter 2. The metric which we chose to compare the algorithms with is that of the Euclidean distance error of the algorithm's estimation with respect to the real jammer location which was used in each scenario.

Like previously mentioned, we used datasets which consist of two different topologies, four different jammer types and 16 different jammer positions in each scenario. This totals to 128 unique scenarios, on which each of the five algorithms was tested. This gives us diversity in our results and has also presented us with the chance to explore a wide range of network conditions/ jammer placements and see how each algorithm behaves under those circumstances.

While it is true that having lots of different outputs can help with generating a lot of conclusions, it might become too much for the reader to see hundreds of tables. Since in each jammer position, there are fourteen batches of 24 nodes, the information will become too overwhelming and difficult to read, had we only noted down the distance errors for each batch. We have compressed this data by only using the average distance error across all the batches for a scenario. To facilitate an understanding of the results, most of them will be presented in the form of graphs with a brief explanation underneath them. This combination of graphical representation plus a small explanation, will help in interpreting the data and provide insights on key findings, patterns, and limitations.

Overall, this chapter is the most crucial component of our thesis, as it focuses on a systematic comparison and aids in giving valuable insights that can contribute to the enhancement of jammer localization techniques.

5.1 Result Examples

To kick off the demonstration of our work, let' see a snapshot of how each dataset was tested. The algorithms were implemented and tested in the Visual Code Studio Editor, which supports python language programs development. It is also very user-friendly, and easy to use, by allowing automation in the project setup.

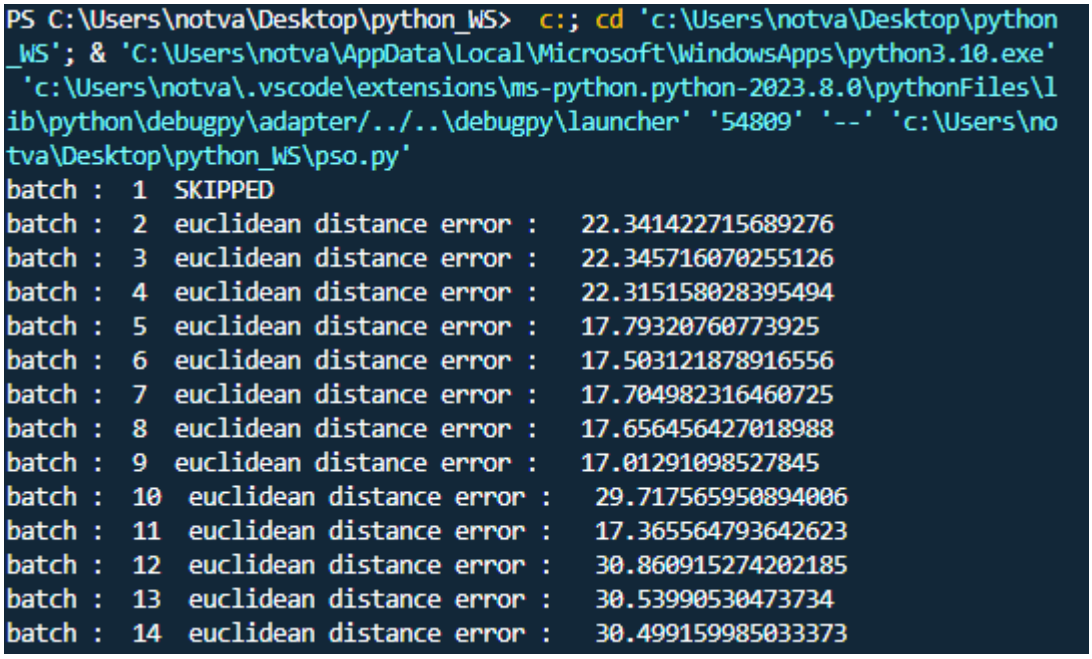


Figure 5.1.1 - Snapshot of Output

Figure 5.1.1 – Snapshot of Output is an example of one of the many executions of our algorithms. More specifically, this snapshot was taken when the PSO algorithm was tested on the randomly distributed topology, when the random jammer was used and placed in the second location of the sixteen different locations (-20, -60). As we can see, the program iterates over the fourteen batches found in the excel file we specify within the code. This is a good time to explain that the readings of the nodes, e.g., the information about which nodes were jammed or not, as well as the information on the coordinates of these nodes are loaded into panda data frames directly from the excel files, to allow for easy processing. In Figure 5.1.1- Snapshot of Output we showcase the output of the PSO algorithm, in which we print the Euclidean distance error, between the algorithm's estimation and the actual jammer's location. You might notice that some batches do not provide with enough information. In this example, the first batch of nodes is skipped, because there were no jammed nodes recorded, therefore the algorithm is unable to localize the jammer, since no attack is recorded.

In the following Figure 5.1.2 – Snapshot of Output 2, we can also see the exact x and y coordinates of each estimation.

```
batch :  1  SKIPPED
batch :  2 has calculated that the position is :  [[  0.00430953 -49.99639833]]
batch :  2  euclidean distance error :   22.366145086598607
batch :  3 has calculated that the position is :  [[ -0.01530987 -50.03832001]]
batch :  3  euclidean distance error :   22.32986582524509
batch :  4 has calculated that the position is :  [[  0.00265189 -50.01697009]]
batch :  4  euclidean distance error :   22.355468433030104
batch :  5 has calculated that the position is :  [[-29.50722665 -45.08089298]]
batch :  5  euclidean distance error :   17.690876543167207
batch :  6 has calculated that the position is :  [[-29.41188681 -44.33473579]]
batch :  6  euclidean distance error :   18.275232309412758
batch :  7 has calculated that the position is :  [[-29.51624022 -45.14437749]]
batch :  7  euclidean distance error :   17.642231947534725
batch :  8 has calculated that the position is :  [[-29.457664   -44.68445815]]
batch :  8  euclidean distance error :   18.000367509169774
batch :  9 has calculated that the position is :  [[-29.51992031 -45.13045857]]
batch :  9  euclidean distance error :   17.655937956591412
batch :  10 has calculated that the position is :  [[-49.63730036 -50.72652286]]
batch :  10  euclidean distance error :   31.054258173605756
batch :  11 has calculated that the position is :  [[-29.78580865 -46.80569952]]
batch :  11  euclidean distance error :   16.427160919781617
batch :  12 has calculated that the position is :  [[-49.44770217 -53.65027445]]
batch :  12  euclidean distance error :   30.124511239273417
batch :  13 has calculated that the position is :  [[-49.59306327 -51.83594916]]
batch :  13  euclidean distance error :   30.69855240440516
batch :  14 has calculated that the position is :  [[-49.48694826 -53.21831913]]
batch :  14  euclidean distance error :   30.25675648459144
```

Figure 5.1.2 - Snapshot of Output 2

The calculated position of the jammer for each batch is displayed above the corresponding Euclidean distance error for each batch. Once again, there are minor differences in the distance errors, even though the scenario remains the same. This is caused by the random nature of the algorithm. To balance these fluctuations, we chose to use the average error resulting from all the fourteen batches. By using the average error resulting from all the batches, we also balance between batches that provide good information and therefore produce a smaller distance error, and batches that provide not that accurate information which leads to bigger distance errors. For example, batch 9 has relatively good information, which is evident by the small distance error (17.65593), whereas batch 13 gives a much bigger error (30.6985), meaning that the recordings on the jamming conditions of the nodes were not as good quality as batch 9. The x and y coordinates of each batch's calculation are not being used in the comparisons, since we only care about the Euclidean distance error to evaluate the performance. By printing them, we can show that the Euclidean distance error is not just a fictional or random generated number, but instead is calculated between the estimation and the real location.

In the following figures, Figure 5.1.3 – Snapshot of output Centroid, Figure 5.1.4 – Snapshot of output weighted Centroid, Figure 5.1.5 – Snapshot of output DCL, Figure 5.1.6 – Snapshot of output VFIL, we showcase the outputs of the other four algorithms discussed in chapter 3.

```
PS C:\Users\notva\Desktop\python_WS>  c:; cd 'c:\Users\notva\Desktop\python_WS'; & 'C:\Users\notva\AppData\Loca
l\Microsoft\WindowsApps\python3.10.exe' 'c:\Users\notva\.vscode\extensions\ms-python.python-2023.8.0\pythonFile
s\lib\python\debugpy\adapter/../..\debugpy\launcher' '55695' '--' 'c:\Users\notva\Desktop\python_WS\ADE\centroi
d_336.py'
 BATCH :  1  skipped
batch :  2  euclidean distance error :   22.360679774997898
batch :  3  euclidean distance error :   22.360679774997898
batch :  4  euclidean distance error :   22.360679774997898
batch :  5  euclidean distance error :   17.755280904564703
batch :  6  euclidean distance error :   17.755280904564703
batch :  7  euclidean distance error :   17.755280904564703
batch :  8  euclidean distance error :   17.755280904564703
batch :  9  euclidean distance error :   17.755280904564703
batch :  10  euclidean distance error :   34.576806613039835
batch :  11  euclidean distance error :   17.755280904564703
batch :  12  euclidean distance error :   34.576806613039835
batch :  13  euclidean distance error :   27.018512172212592
batch :  14  euclidean distance error :   27.018512172212592
```

Figure 5.1.3 – Snapshot of Output Centroid

```
PS C:\Users\notva\Desktop\python_WS>  c:; cd 'c:\Users\notva\Desktop\python_WS'; & 'C:\Users\notva\AppData\Loca
l\Microsoft\WindowsApps\python3.10.exe' 'c:\Users\notva\.vscode\extensions\ms-python.python-2023.8.0\pythonFile
s\lib\python\debugpy\adapter/../..\debugpy\launcher' '55725' '--' 'c:\Users\notva\Desktop\python_WS\ADE\weighte
d-centroid-336.py'
SKIPPED BATCH :  1
22.360679774997898
22.360679774997898
22.360679774997898
17.755280904564703
17.755280904564703
15.045561866135618
15.045561866135618
17.755280904564703
28.159562990795276
17.755280904564703
34.57680661303984
23.101843804839525
28.083140248028414
```

Figure 5.1.4 – Snapshot of Output Weighted Centroid

```
PS C:\Users\notva\Desktop\python_WS>  c:; cd 'c:\Users\notva\Desktop\python_WS'; & 'C:\Users\notva\AppData\Loca
l\Microsoft\WindowsApps\python3.10.exe' 'c:\Users\notva\.vscode\extensions\ms-python.python-2023.8.0\pythonFile
s\lib\python\debugpy\adapter/../..\debugpy\launcher' '55827' '--' 'c:\Users\notva\Desktop\python_WS\ADE\circles
-336.py'
SKIPPED BATCH :  1
SKIPPED BATCH :  2  NOT ENOUGH NODES TO MAKE A HULL
SKIPPED BATCH :  3  NOT ENOUGH NODES TO MAKE A HULL
SKIPPED BATCH :  4  NOT ENOUGH NODES TO MAKE A HULL
SKIPPED BATCH :  5  NOT ENOUGH NODES TO MAKE A HULL
SKIPPED BATCH :  6  NOT ENOUGH NODES TO MAKE A HULL
SKIPPED BATCH :  7  NOT ENOUGH NODES TO MAKE A HULL
SKIPPED BATCH :  8  NOT ENOUGH NODES TO MAKE A HULL
SKIPPED BATCH :  9  NOT ENOUGH NODES TO MAKE A HULL
Batch 10 - Distance error:  31.969370830826037
SKIPPED BATCH :  11  NOT ENOUGH NODES TO MAKE A HULL
Batch 12 - Distance error:  31.969370830826037
Batch 13 - Distance error:  31.96937083082605
Batch 14 - Distance error:  31.969370830826023
```

Figure 5.1.5 – Snapshot of Output DCL

Notice that there are some batches that do not provide sufficient information for the DCL algorithm to function normally. As discussed in 3.1.3, the DCL algorithm requires at least three jammed nodes to function, because a convex hull cannot be created with two or less points. We ignore batches that contain insufficient information and only process the ones that will lead to estimations. This is a huge downside in DCL, because the information comes from simulating realistic scenarios, meaning that DCL will malfunction in some real-world situations.

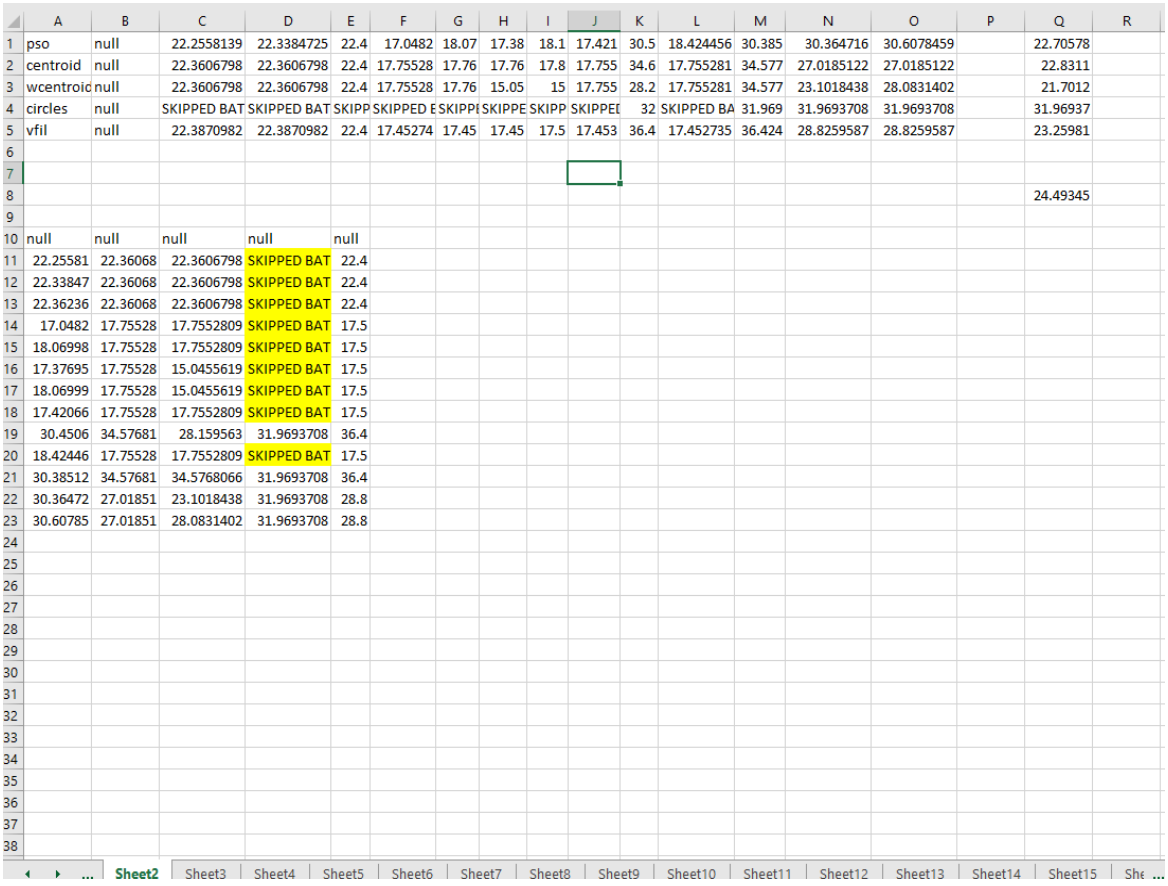
```
PS C:\Users\notva\Desktop\python_WS>  c:; cd 'c:\Users\notva\Desktop\python_WS'; & 'C:\Users\notva\AppData\Loca
l\Microsoft\WindowsApps\python3.10.exe' 'c:\Users\notva\.vscode\extensions\ms-python.python-2023.8.0\pythonFile
s\lib\python\debugpy\adapter/../..\debugpy\launcher' '55872' '--' 'c:\Users\notva\Desktop\python_WS\ADE\VFIL-33
6.py'
node_batch ID :  1 SKIPPED
Batch num :  2 The error rate is: 22.3870981577124
Batch num :  3 The error rate is: 22.3870981577124
Batch num :  4 The error rate is: 22.3870981577124
Batch num :  5 The error rate is: 17.452735247640717
Batch num :  6 The error rate is: 17.452735247640717
Batch num :  7 The error rate is: 17.452735247640717
Batch num :  8 The error rate is: 17.452735247640717
Batch num :  9 The error rate is: 17.452735247640717
Batch num :  10 The error rate is: 36.42397307900464
Batch num :  11 The error rate is: 17.452735247640717
Batch num :  12 The error rate is: 36.42397307900464
Batch num :  13 The error rate is: 28.82595867208444
Batch num :  14 The error rate is: 28.82595867208444
```

Figure 5.1.6 – Snapshot of output VFIL

This is how we tested each algorithm for all the possible scenarios. This information is stored across multiple excel files, which will be explained in the next subchapter of this paper.

5.2 Storing and Processing the Results

Due to the overwhelming amount of information that each simulation provides, we had to come up with a way to store and organize our data that would allow for easy processing. We did that by creating an excel file for each jammer location per topology. In other words, there is an excel file with 16 sheets, per topology type per jammer, with each sheet storing the calculations of the five algorithms for one of the sixteen possible jammer locations.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	pso	null	22.2558139	22.3384725	22.4	17.0482	18.07	17.38	18.1	17.421	30.5	18.424456	30.385	30.364716	30.6078459		22.70578	
2	centroid	null	22.3606798	22.3606798	22.4	17.75528	17.76	17.76	17.8	17.755	34.6	17.755281	34.577	27.0185122	27.0185122		22.8311	
3	wcentroid	null	22.3606798	22.3606798	22.4	17.75528	17.76	15.05	15	17.755	28.2	17.755281	34.577	23.1018438	28.0831402		21.7012	
4	circles	null	SKIPPED BAT	SKIPPED BAT	SKIPP	SKIPPED E	SKIPP	SKIPPE	SKIPP	SKIPPE	32	SKIPPED BA	31.969	31.9693708	31.9693708		31.96937	
5	vfil	null	22.3870982	22.3870982	22.4	17.45274	17.45	17.45	17.5	17.453	36.4	17.452735	36.424	28.8259587	28.8259587		23.25981	
6																		
7																		
8																	24.49345	
9																		
10	null	null	null	null	null													
11	22.25581	22.36068	22.3606798	SKIPPED BAT	22.4													
12	22.33847	22.36068	22.3606798	SKIPPED BAT	22.4													
13	22.36236	22.36068	22.3606798	SKIPPED BAT	22.4													
14	17.0482	17.75528	17.7552809	SKIPPED BAT	17.5													
15	18.06998	17.75528	17.7552809	SKIPPED BAT	17.5													
16	17.37695	17.75528	15.0455619	SKIPPED BAT	17.5													
17	18.06999	17.75528	15.0455619	SKIPPED BAT	17.5													
18	17.42066	17.75528	17.7552809	SKIPPED BAT	17.5													
19	30.4506	34.57681	28.159563	31.9693708	36.4													
20	18.42446	17.75528	17.7552809	SKIPPED BAT	17.5													
21	30.38512	34.57681	34.5768066	31.9693708	36.4													
22	30.36472	27.01851	23.1018438	31.9693708	28.8													
23	30.60785	27.01851	28.0831402	31.9693708	28.8													
24																		
25																		
26																		
27																		
28																		
29																		
30																		
31																		
32																		
33																		
34																		
35																		
36																		
37																		
38																		

Sheet2 Sheet3 Sheet4 Sheet5 Sheet6 Sheet7 Sheet8 Sheet9 Sheet10 Sheet11 Sheet12 Sheet13 Sheet14 Sheet15

Figure 5.2.1 – Example of an Excel Workbook

In the above Figure 5.2.1 - Example of an Excel Workbook, we can see how the results of the five algorithms shown in 5.1 were stored. There are 16 total sheets, with each sheet storing the information on the corresponding jammer position. Sheet 2 is selected, meaning that we

are seeing the results from the second jammer position (-20, -60), when a randomly distributed topology was used alongside a random jammer. The cells in the range A1:A5 hold the name of each algorithm. Then, each column from B to O has the Euclidean distance error for each batch, hence there are 14 columns to represent the 14 batches. At column Q, we store the average error for that algorithm which is also the information we use later in the comparison of the algorithms. By storing the average error, we can use it to see how each algorithm performs across different topologies, different jammer types and different jammer locations. Lastly, at the cell Q8 we hold the average error from all the algorithms. Once again, this is useful in pointing out how different combinations of topology type, jammer type and jammer location generally affect the performance of the five algorithms.

After all 16 possible jammer locations, shown in Figure 2.2.5 – Jammer Coordinates have been tested, we proceed to concatenate the information from each batch and store them in another excel file. This Excel file calculates the averages of the five algorithms plotted to the topology type, jammer type and jammer placement. In the following Figure 5.2.2 – Snapshot of Data, we can see the structure of the file.

A	B	C	D	E	F	G	H	I
Avg error per scenario				PSO	centroid	wcentroid	circles	vfil
17.46222628				16.48639	16.12452	17.14356	18.12741	19.42926
24.49345485				22.70578	22.8311	21.7012	31.96937	23.25981
16.29846006				18.01254	16.70178	16.70178	11.79357	18.28264
25.67984836				26.57566	25.5718	25.57975	21.73151	28.94053
17.31529463				19.97243	17.87841	17.87841	6.940519	23.90671
24.09406107				26.61932	26.7244	26.66294	12.79368	27.66997
17.36610836				20.02191	19.54132	19.54132	9.469041	18.25695
31.27580107				31.98753	31.40292	31.02858	33.43056	28.52942
26.46612802				26.65545	26.65427	26.65427	no predict	25.90053
15.32328923				16.29309	15.46922	15.49304	14.25398	15.10711
16.16104945				19.33566	14.74745	14.75723	19.94334	12.02156
24.44459711				27.87623	28.84379	28.96297	11.67457	24.86543
11.62686667				13.66852	11.77919	11.73923	9.517894	11.4295
40.41528389				53.72468	34.28451	38.67212	39.79113	35.60399
22.32663895				23.33938	23.68081	23.71238	18.1113	22.78932
25.27530803				29.13274	22.39289	22.43139	30.02768	22.39184
				24.52546	22.16427	22.41626	19.30504	22.39904

Figure 5.2.2 - Snapshot of Data

We have organized the results like this: At the columns from E to I we store the average error of each algorithm. Each row represents a jammer location, hence the 16 total rows. This means that under each algorithm's name, we see the average Euclidean distance error from the 14 batches of that algorithm in each jammer location. Also, we calculate the average error of all the algorithms per jammer location in the column A. By doing this, we get an idea of how the jammer placement affects the algorithms in a more general way. At the bottom of the snapshot, we can see the average Euclidean distance error for each algorithm across all 16 jammer placements in the topology. Once again this sums the performance of each algorithm and gives us a clearer picture on the performance of each algorithm. Here, one simple deduction which occurs simply by looking at the numbers, is that PSO has underperformed in the combination of randomly distributed topology with a random jammer, averaging an error of 24.52546 across the 16 possible jammer locations. Moreover, we can also infer that when the jammer was placed in the 14th position (-20, 60), it causes the biggest error across all five algorithms, reaching a value of 40.415284. Lastly, the different colours visible in the column concerning the DCL algorithm, are used to show how much the average error was affected by the lack of DCL to provide an estimation in some batches. A yellow colour indicates that there isn't much divergence compared to the average error of other algorithms and the possible average error of DCL had the algorithm given an estimated position in all batches. An orange colour shows that the average error is higher than expected, because some batches that give low average errors for the other four algorithms cannot be of use from DCL, therefore its average error was calculated from relatively poor information. A red colour indicates that there is huge difference in DCL's average error, because some poor batches have not been included in the calculations. In other words, the average error resulting from DCL can be misleading in some cases, meaning that the average error across all jammer positions (19.30504) for this combination of jammer type plus topology does not reflect the performance of DCL in the same level of accuracy compared to the other four algorithms. Lastly, in the scenario in which the jammer was placed in the 9th position, DCL completely failed to provide an estimation throughout the 14 batches, which would have a catastrophic result in a real-life scenario.

In this same excel file, we have plenty of graphs that help in understanding the data, by visualising the performance of the algorithms plotted to various variables. More information is given under each graph in the next section.

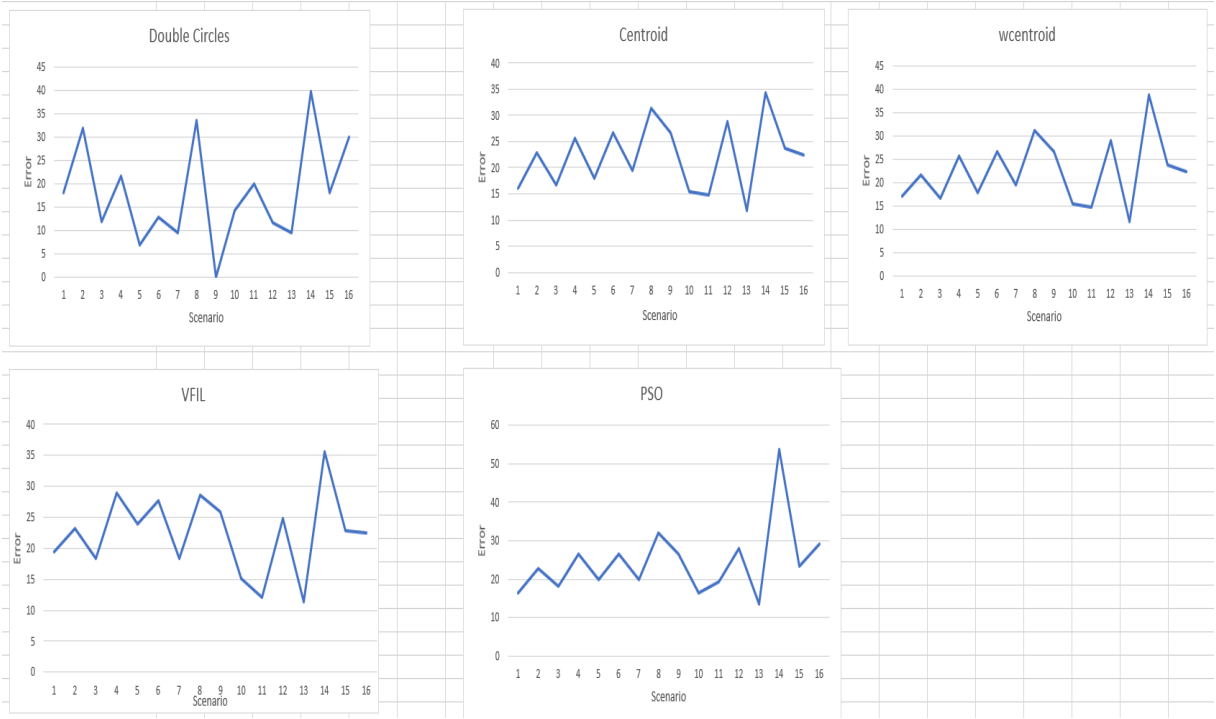


Figure 5.2.3 – Performance of Algorithms

In figure 5.2.3 - Performance of Algorithms, we can see the average error of the five algorithms, based on the location that the jammer was placed. It is evident that the algorithms follow the same trends, and we can observe that they have very similar errors across the different jammer locations. DCL seems to be the best algorithm, but this is not the case, as there are misleading results in some jammer locations which cause a smaller average error. Overall, none of the algorithms is doing particularly well when a random jammer was used in a topology with randomly distributed nodes. We can interpret this observation in many ways:

1. When the jammer was placed at the bottom right of the topology, it caused a huge error across all the algorithms. If we look at the topology shown in Figure 2.2.2 – Random Distribution Topology, we can see that there are not that many nodes scattered near that location of the jammer, meaning that the information we gave our algorithms was of relatively bad quality.
2. PSO algorithm, has performed poorly, with respect to the other algorithms and our expectations. We did not expect this, since the algorithm generates particles across the network, it would be able to handle situations like this better than the other algorithms who greatly benefit by a uniform node distribution and good node

density. This means that PSO is heavily relying on the quality of the data. When the data is not of particularly good quality, the algorithm underperforms.

3. DCL algorithm is not suitable for this topology, as it misses a lot of estimations due to the lack of sufficient jammed nodes in most of the batches.
4. CL, WCL and VFIL perform almost identically, with a relatively average performance. Since these three algorithms follow similar approaches, we can say that they are bound by the quality of the data. The approach of calculating the centroid and then fixing the calculation notwithstanding, VFIL does not improve the localization accuracy.

We further analyse the results in the excel file by providing more graphs.

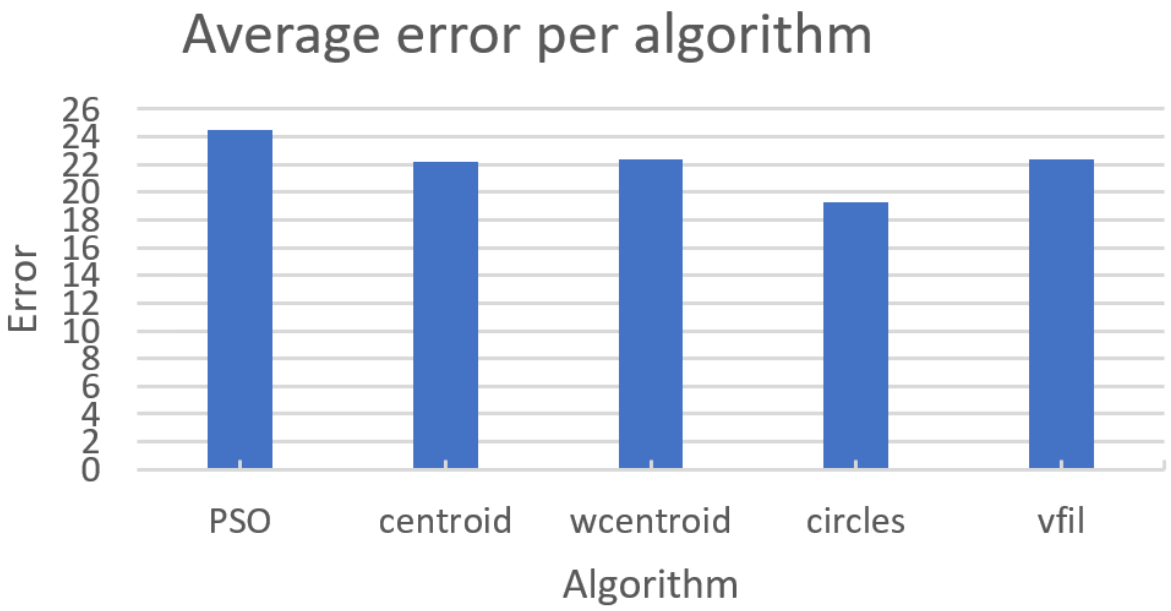


Figure 5.2.4 Average error Per Algorithm

In Figure 5.2.4 - Average error Per Algorithm we can see the average error of the algorithms, in a clearer way. Our conclusions are verified by seeing that the algorithms perform similarly. Moreover, we can safely say that all algorithms give a relatively big average error, which means that:

1) The random distribution topology when a random jammer is used is a big challenge in general.

2) The data we were supplied with is not good enough to provide the algorithms with information that gives good localization accuracy.

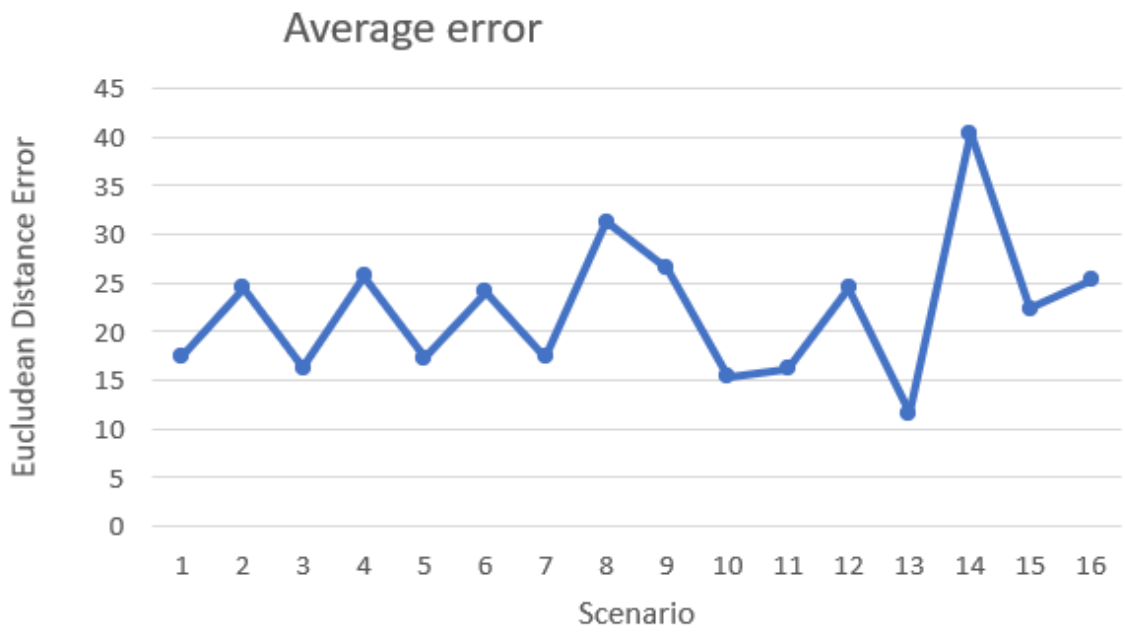


Figure 5.2.5 – Average error per scenario

In Figure 5.2.5 - Average error per scenario, we show the graph in which we plotted the average Euclidean Distance Error to the jammer location. We can see that apart from the 14th location which has given a huge average distance error, there is no clear indication on how the jammer location affects the performance of the algorithms, since there are continuous ups and downs across different locations. This is how we evaluate the performance of each algorithm, across different topologies and jammer types. By recording the information and grouping it like this, we can analyse the effects of the different conditions of the simulation on the localization error. In the following section, we will provide the graphs from

experiments from which we deducted the insights that will be discussed at the end of this chapter. All the graphs are followed by a brief explanation, and some deductions.

5.3 Result Analysis

In this subchapter, we present the most relevant graphs from each simulation, along with a brief yet concise explanation about the scenario and some conclusions. As previously mentioned, we will not provide the results in the same level of detail as in 5.2, since it will be hard for the reader to find them meaningful, whereas a condensed representation of the results will avoid overwhelming the reader.

5.3.1 Constant Jammer

5.3.1.1 Constant Jammer on uniformly distributed topology

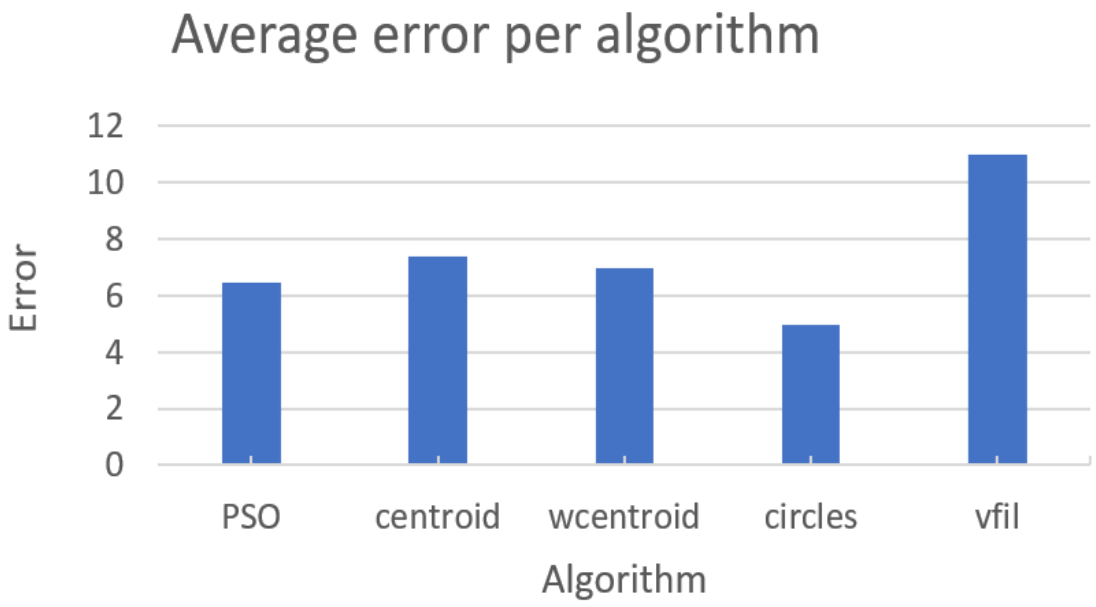


Figure 5.3.1.1.1

Figure 5.3.1.1.1 illustrates the performance of the algorithms in the scenario where a constant jammer was used in a uniformly distributed topology. From the graph, we can

see that PSO algorithm outperforms the rest of the algorithms. While the DCL algorithm (circles) seems to be having a lower average, this doesn't accurately reflect the reality, since some batches were excluded from the calculations because not enough data was available for DCL to provide an estimation. Since a constant jammer was used, which is the easiest to detect as we discussed in 4.1.1, we have very low localization errors, indicating the high quality of the data.

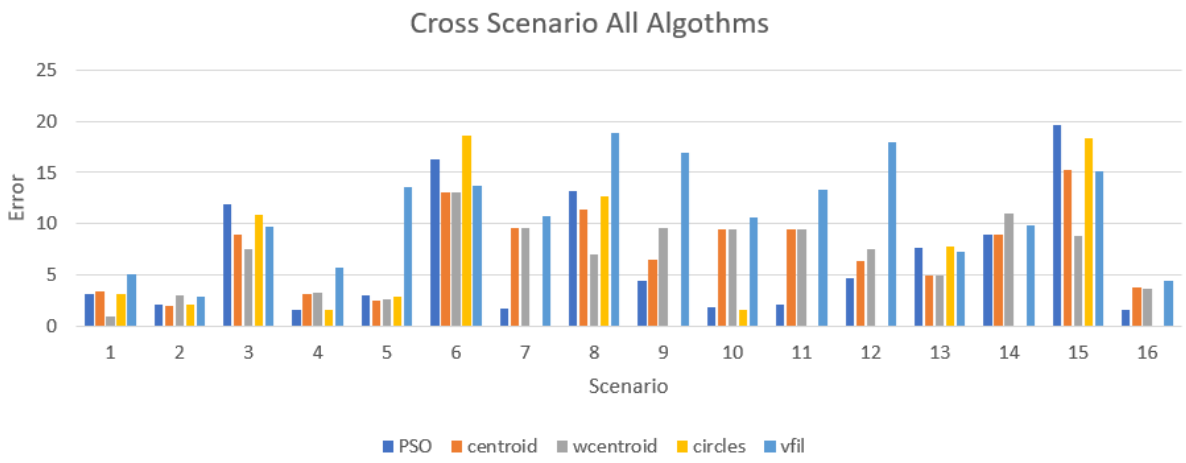


Figure 5.3.1.1.2

Figure 5.3.1.1.2 presents the performance of the algorithms with regards to each jammer position (X axis). Once again, we can see that the placement of the jammer makes no difference which is expected since the nodes in the topology are distributed uniformly. Notably, the readings from the fuzzy logic detection system are pretty much the same for all positions, leading to similar performances across the algorithms.

5.3.1.2 Constant Jammer on randomly distributed topology

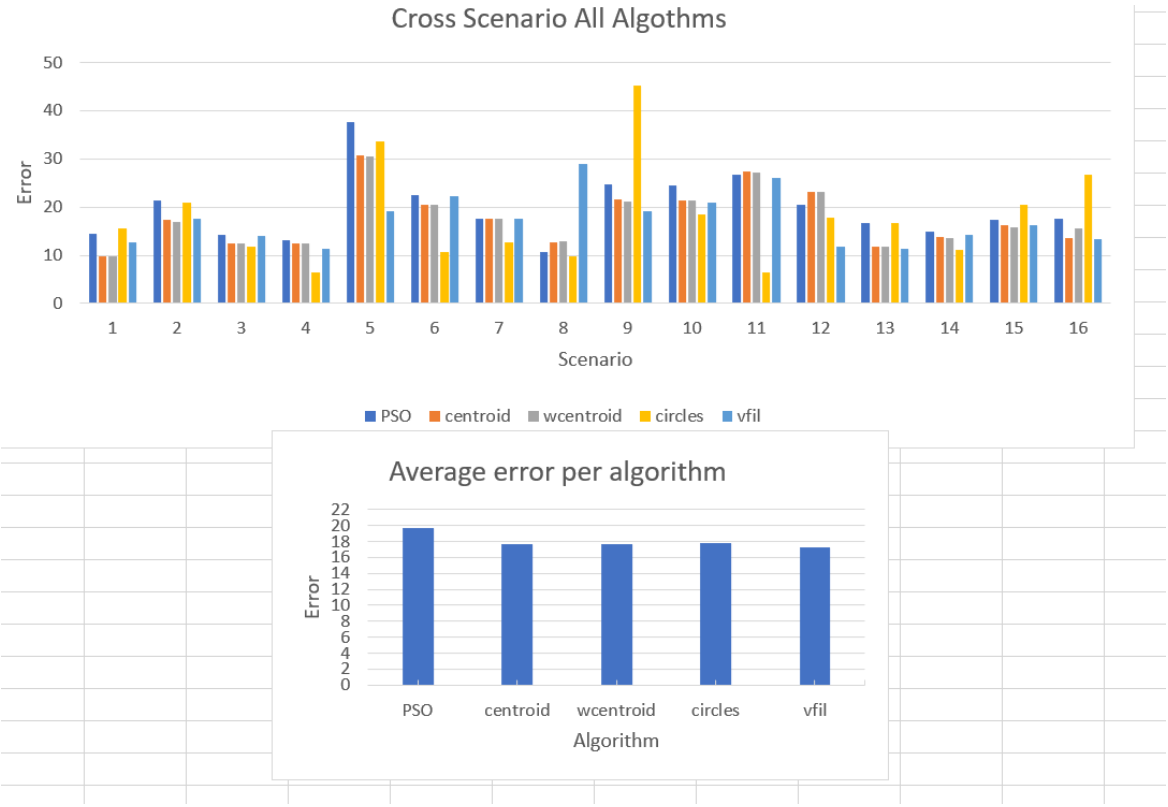


Figure 5.3.1.2.1

Figure 5.3.1.2.1 demonstrates the immediate effect of the topology type on the performance of the algorithms when the same type of jammer as previously was used. While the jammer type and jammer positions remain the same, transitioning from a uniform to a random topology has a tremendous impact on the localization error. All five algorithms perform poorly and have a localization error twice as much as what they had in the uniformly distributed topology.

5.3.2 Deceptive Jammer

5.3.2.1 Deceptive Jammer on uniformly distributed topology

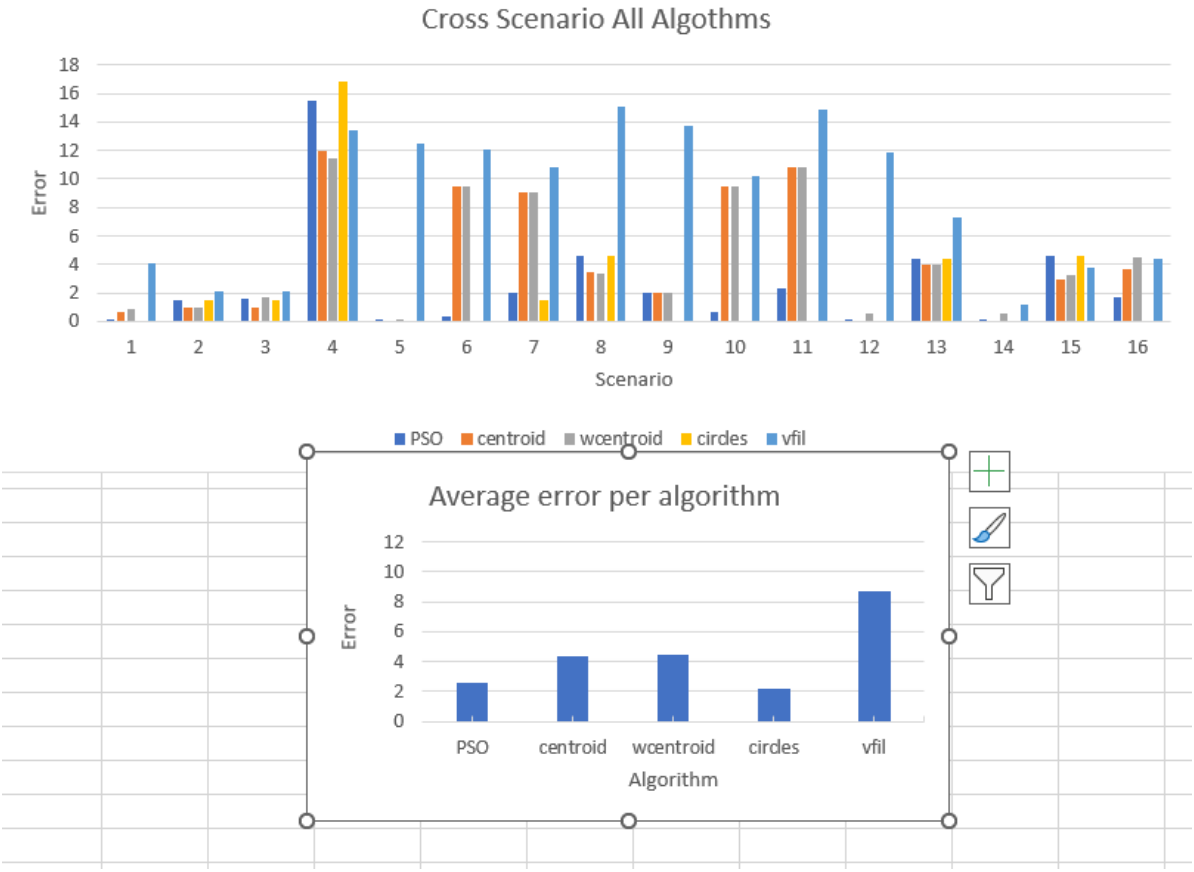


Figure 5.3.2.1.1

In Figure 5.3.2.1.1, where a deceptive jammer was used, we observe that all algorithms, except VFIL, exhibit improved performance compared to the scenario where a constant jammer was used. PSO excels in this scenario, showing the capabilities of the algorithm we implemented when the readings of the datasets have good quality. So far, we can say that the jammer type has almost no difference in the performance of PSO since both jammer types have produced similar errors. Additionally, jammer positions closer to the sink (e.g., 6,7,10,11) result in higher errors, something that does not occur when a constant jammer is used.

5.3.2.2 Deceptive Jammer on randomly distributed topology

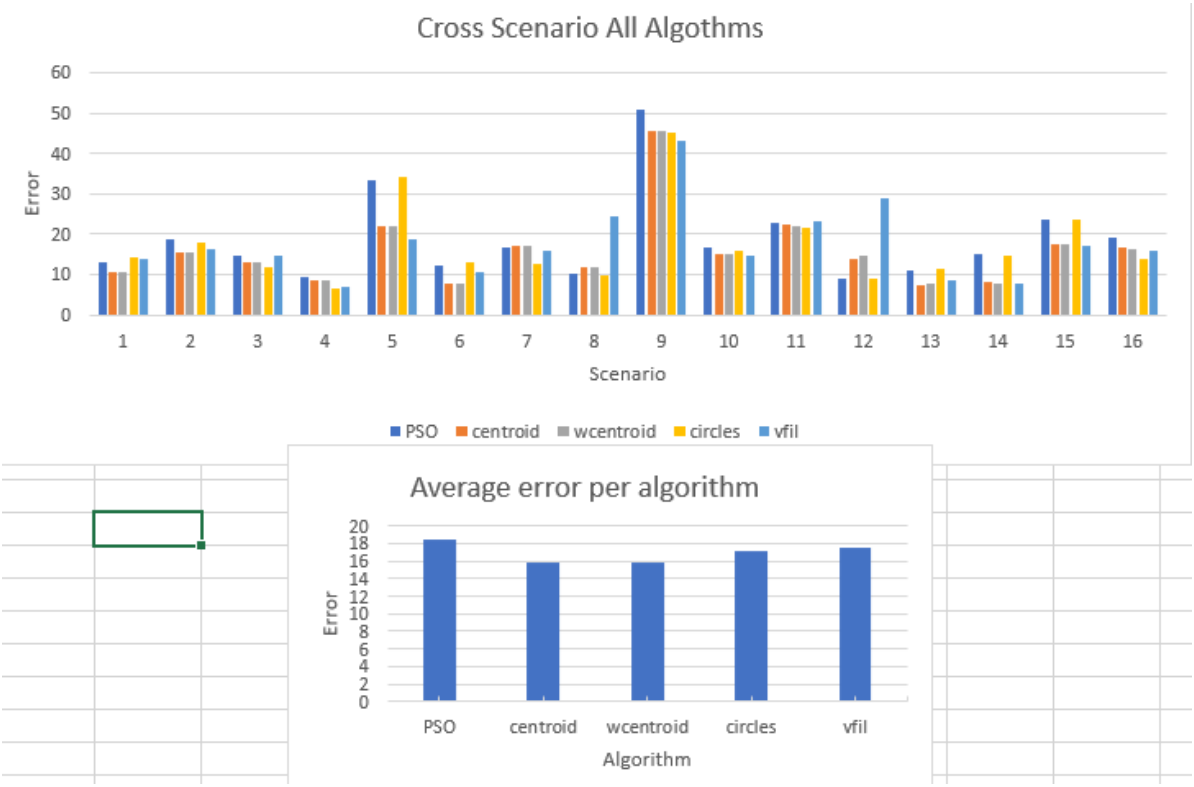


Figure 5.3.2.2.1

Figure 5.3.2.2.1 confirms the negative impact from changing the topology from uniform to random on both five algorithms, and especially PSO since it has the biggest difference between the average errors across the two topology types. Moreover, jammer position plays no role in the average error across all algorithms. Interestingly, there is very little difference between the errors that are caused by the constant jammer and the deceptive jammer in randomly distributed topologies, unlike the more significant disparity observed when a uniform topology is used.

5.3.3 Random Jammer

5.3.3.1 Random Jammer on uniformly distributed topology

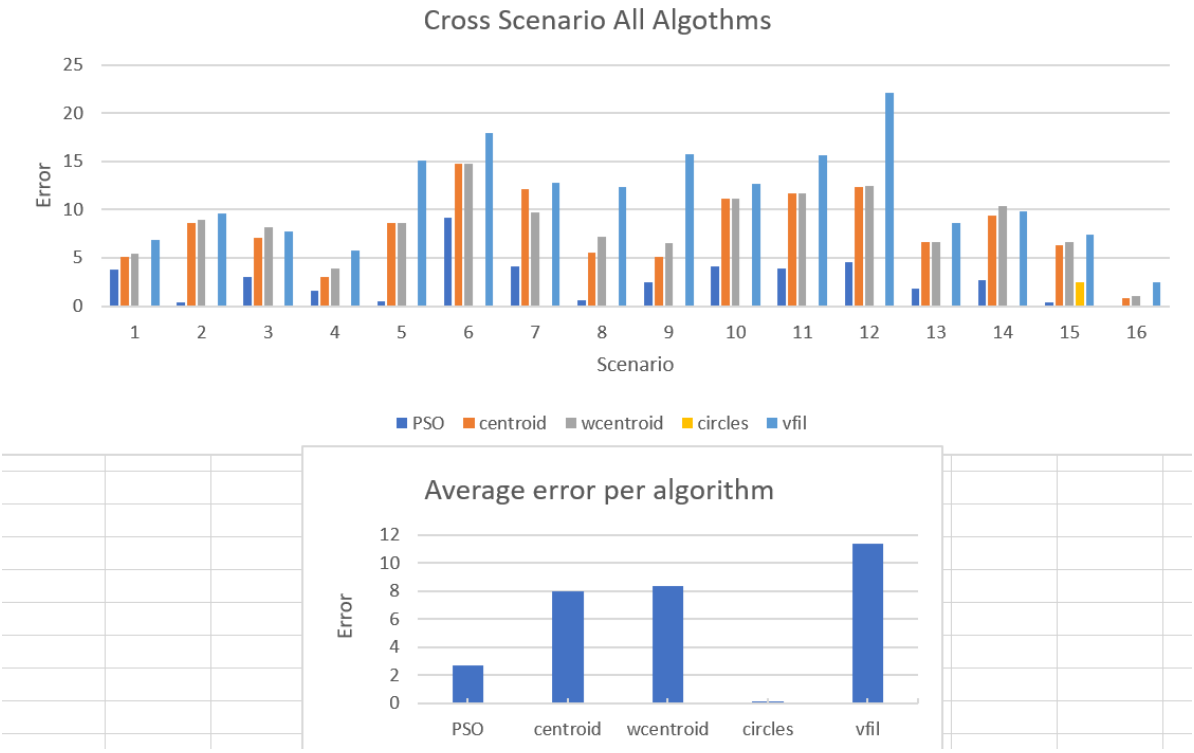


Figure 5.3.3.1.1

Figure 5.3.3.1.1 displays the performance of the algorithms when a random jammer was used on a uniformly distributed topology PSO and DCL once again demonstrate their extreme efficiency in localizing the jammer, while the other three algorithms perform at an average level, similar to their performances in a uniformly distributed topology for constant and deceptive jammers. It is important to note that DCL fails to provide an estimation for some batches, indicating that its average error in a real-life scenario would be higher than the one shown in graphs. Furthermore, jammer position does not affect the performance of any of the 5 algorithms.

5.3.3.2 Random Jammer on randomly distributed topology

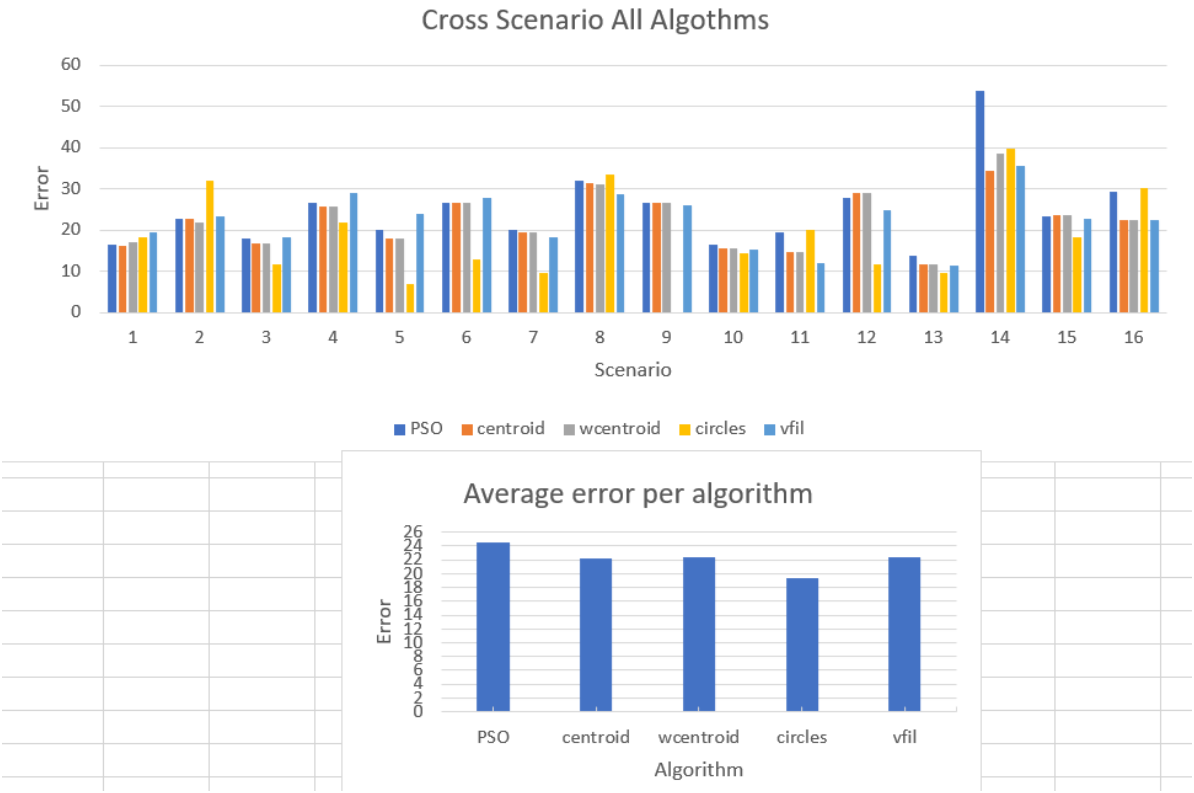


Figure 5.3.3.2.1

In Figure 5.3.3.2.1, it is clearly visible that when a random jammer was used on a randomly distributed topology, we had the biggest average errors out of all other jammer/topology combinations. None of the algorithms has successfully managed to localize the jammer in any jammer location. This very likely suggests the fuzzy index readings of the nodes in this simulation are of poor quality and this is evident in the significantly higher localization errors. Since this experiment aligns with the one used in 5.2 to demonstrate the results, there is no need for further analysis.

5.3.4 Reactive Jammer

5.3.4.1 Reactive Jammer on uniformly distributed topology

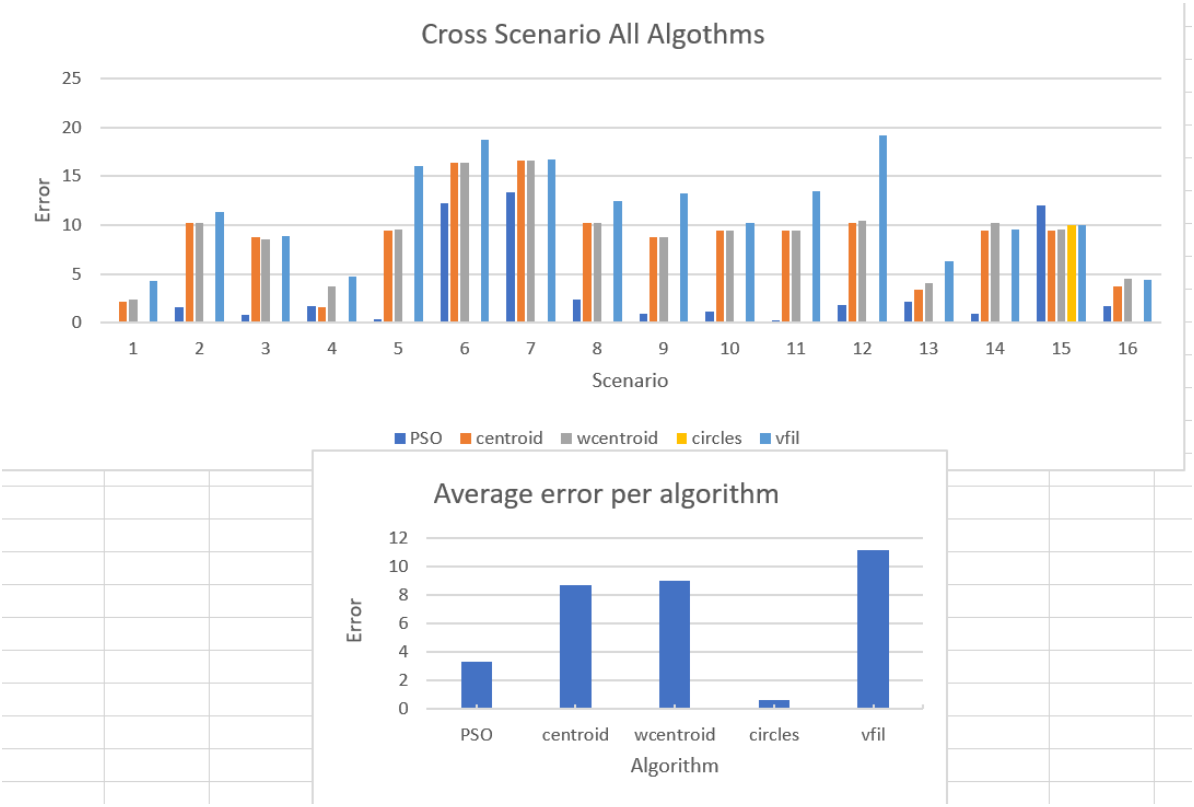


Figure 5.3.4.1.1

Figure 5.3.4.1.1 visualises the effects of using a reactive jammer in a uniformly topology. Much like all the other types of jammers in this type of topology, we have relatively low localization errors. For once more, PSO is doing exceptionally good in localizing the jammer, with DCL yielding similar results. However, both the CL and WCL algorithms, with this type of jammer exhibit slightly poorer performance than anticipated, much like their performance with a random jammer. Jammer location has no effect on the performance of the algorithm.

5.3.4.2 Reactive Jammer on uniformly distributed topology

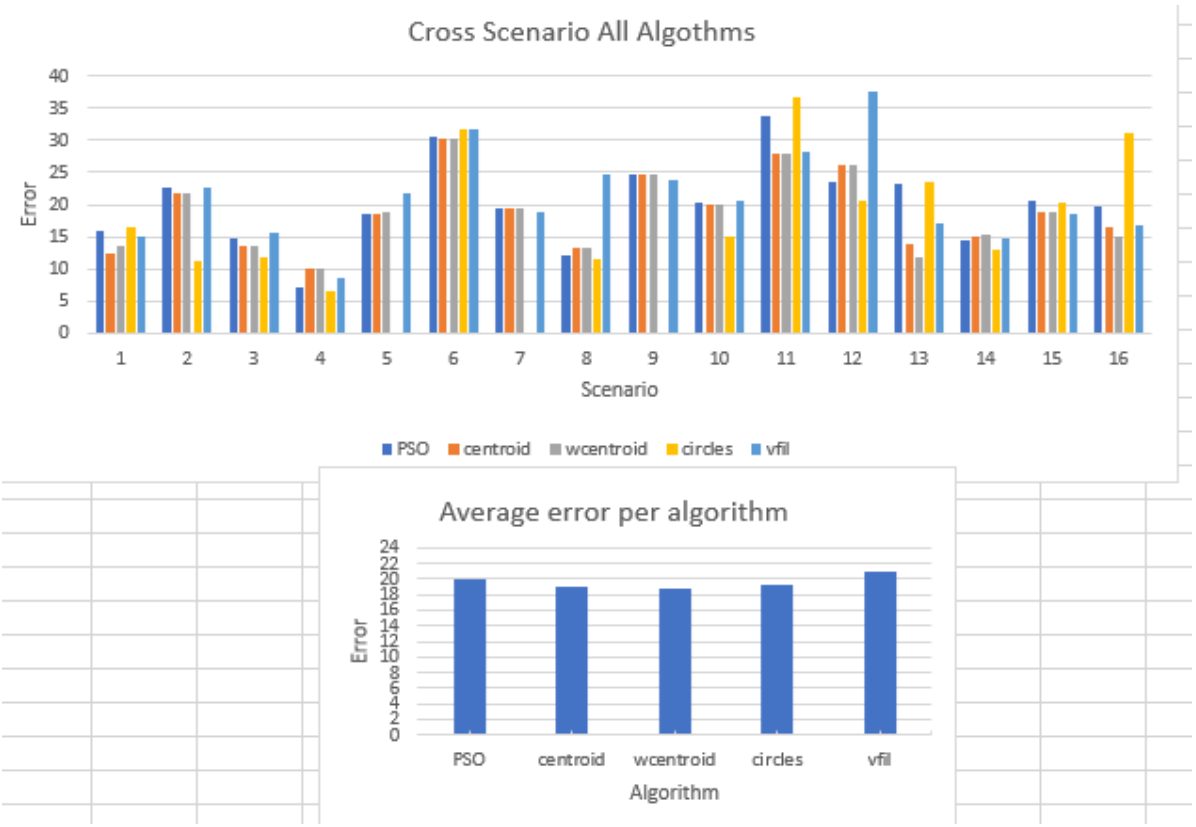


Figure 5.3.4.2.1

Lastly, Figure 5.3.4.2.1 depicts the performance of each algorithm when a reactive jammer was used in a randomly distributed topology. Similar to previous experiments with random topologies, there is a significant difference in the performance of all algorithms compared to their performance with the same jammer type but in a uniformly distributed topology. Notably, DCL totally fails to provide an estimation for more jammer locations in this topology compared to other topology/jammer type simulations. It is once again obvious that the performance of the algorithms is constrained by the quality of the provided data. This can be deducted from the fact that PSO, which as a concept follows a completely different approach than the other four algorithms, gives almost identical localization errors with the other four algorithms.

By summarizing the results of running the algorithms across the eight different combinations of jammer type/topology type, we can see the effect of these parameter on the performance of each algorithm. Individually, these can provide with meaningful insights regarding the performance of each algorithm in each situation. In the following chapter, we provide the results from putting together all the above, in an attempt to gain a broader perspective of our work.

5.4 Overall Performance Analysis and Conclusions

In this subchapter, our aim is to provide a holistic view of how the overall performance of the algorithms is affected by various factors such as jammer type, topology type and jammer location. By combining the results explained in 5.3, we present a comprehensive analysis of the overall work done, to help draw more meaningful and accurate deductions.

5.4.1 Impact of Jammer and Topology Type on Overall Performance

One very important aspect of the work is how different jammers across the two different types of topologies affect the overall performance of the algorithms. To do that we have combined the results and isolated the average error of each algorithm, per jammer type and topology type. This means that we have taken the average errors from each algorithm and separated them depending on the jammer type and the topology.

J	K	L	M	N	O	P	Q
	Uniform Topology	Uniform Topology	Uniform Topology	Uniform Topology	Uniform Topology		
	PSO	centroid	wcentroid	circles	vfil		
Constant	6.457676595	7.390712024	6.935694599	4.95432606	10.97026422		7.341735
Deceptive	2.585258036	4.343679878	4.50234004	2.190054944	8.706640414		4.465595
Random	2.713263824	8.011242576	8.337435336	0.15625	11.41447746		6.126534
Reactive	3.329995821	8.695202017	8.995398618	0.625	11.20622795		6.570365
Topology Average	3.771548569	7.110209124	7.192717148	1.981407751	10.57440251		
	Random topology	Random topology	Random topology	Random topology	Random topology		
	PSO	centroid	wcentroid	circles	vfil		
Constant	19.6555983	17.64164172	17.66570304	17.80278995	17.29201385		18.01155
Deceptive	18.48156554	15.77746642	15.7765592	17.16169262	17.47120301		16.9337
Random	24.52545596	22.16427341	22.41625976	19.3050369	22.39903578		22.16201
Reactive	20.08394754	18.93457255	18.79562716	19.15063336	20.99978945		19.59291
Topology Average	20.68664184	18.62948853	18.66353729	18.35503821	19.54051052		
Total Average	12.2290952	12.86984882	12.92812722	10.16822298	15.05745652		

Figure 5.4.1.1 – Effect of jammer type across algorithms

Figure 5.4.1.1 - Effect of jammer type across algorithms showcases how each algorithm behaved with respect to the four jammer types. For example, PSO averaged a Euclidean distance error of 6.45767 when the constant jammer was used in the uniformly distributed topology. Similarly, for the same jammer type in the randomly distributed topology, it averaged 19.65559. The same goes for the rest of the algorithms. By looking at the data, we can see that in the uniform topology, PSO has clearly outperformed the rest of the algorithm. The constant jammer has caused the worse performance for PSO, while the rest of the jammers have given very low localization errors, compared to the one caused by the constant jammer. By also looking at column Q, which resembles the average error for each jammer type across all algorithms, we can see that the constant jammer has caused the largest average error of 7.34173. This means that if we were in a uniformly distributed topology, the worst type of jammer to face would be the constant jammer, which is surprising since this type of jammer is the least harmful of all. In contrast with this, the deceptive jammer has caused the least error with a value of 4.46559. Apart from the deceptive jammer's relatively low localization error, all the other jammers cause similar errors. From this we can also deduce that the uniformly distributed topology provides with good quality information, which in turn causes our algorithms to perform fairly well even when different types of jammers are used.

We can also see that CL and WCL perform almost identically. This is something that we expected since we set the fuzzy index of each node to be used in WCL as the weight. This means that WCL is only working with the same information that CL does, without taking advantage of other metrics, such as the distance between the centroid, which is something that would provide better localization accuracy.

DCL is performing very good as well, however like previously mentioned, some results are not that accurate due to the miss of some estimations across different jammer locations. Random and reactive types of jammers cause the most misses to DCL, hence the relatively low and misleading localization errors under DCL.

VFIL is severely underperforming in all jammer types when the simulations included a uniformly distributed topology with an average error of 10.5744, meaning that it is the least preferred algorithm to be used in this type of topology. This is most likely related to the

implementation of the algorithm. Since we were supplied with a baseline implementation of the original VFIL algorithm and not with the version proposed in the literature, there could be issues with it. In normal circumstances, VFIL should perform similarly or better than CL and WCL.

Moving on to the results for the randomly distributed topology, the impact of transitioning to this type of topology is evident in the huge difference between average localization errors of the algorithm. PSO performs the worse, with an average error of 20.686641. This has led us to the inclination that we should avoid using PSO when our problem consists of a randomly distributed topology, no matter what type of jammer is used. This is mainly caused by two reasons: 1) The algorithm is heavily relying on good quality information. Since the recordings in the random topology are not that good compared to the recordings of the uniform topology, the algorithm performs badly. In other words, good information provides optimal results, whereas bad quality information is producing worse results than the other four algorithms. 2) There is something wrong with the implementation. PSO is a complicated algorithm and uses a lot of different parameters. It is possible that some parameters do not have the optimal values, therefore the results are suboptimal.

Contrary to the effect of different types of jammers when a uniformly distributed topology was used, in a randomly distributed topology, there are big differences in the localization errors for each jammer. Deceptive jammer is still the jammer that causes the smallest localization error, meaning that in a real-life situation, the best-case scenario will include a deceptive jammer. Constant jammer is causing the second lowest average error, in contrast to the highest average error caused in the uniformly distributed topology. Reactive jammer is causing the second highest error, much like in the uniformly distributed topology in which again reactive jammer comes second in average error. From this we can understand that the algorithms have a stable performance for both topologies when a reactive jammer was used, meaning that this type of jammer is particularly effective across different types of topologies. The random jammer has caused the greatest average error, with a value of 22.162. The effects of a jamming attack when a random jammer is used in a randomly distributed topology are catastrophic, as all five algorithms produce high localization errors. Moreover, the random jammer records the biggest difference between the average error of the uniform topology and the random topology.

CL and WCL are performing identically, which confirms our speculations on why this is happening as mentioned above. Regarding VFIL, it performs slightly better than the PSO, with an average error of 19.5405, which falls short to the performance of CL, WCL and DCL who have errors ~18.5. This means that VFIL, with the variation used in our experiments, should be avoided for both topologies and for all types of jammers. Moreover, random and reactive jammer types cause a lot of misses to DCL when the random topology is used.

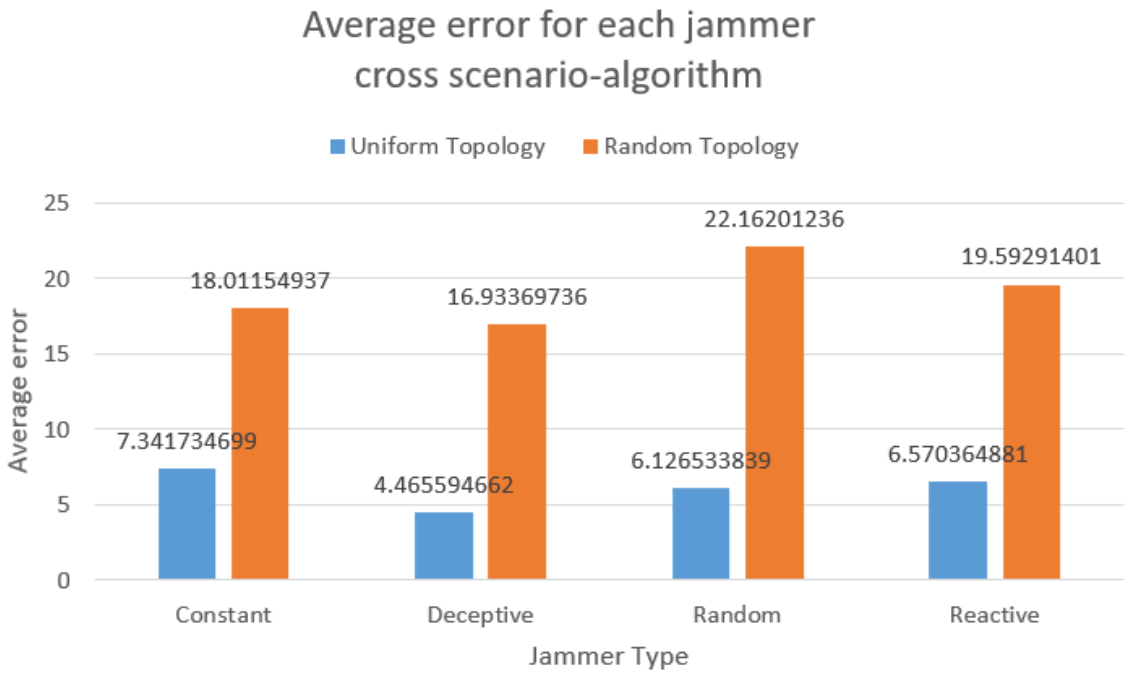


Figure 5.4.1.2

Figure 5.4.1.2 shows a graph which compares the jammers with one another. We can see the average error that each jammer causes for each of the two topology types. This graph summarizes what we mentioned above, by showing that the random jammer has the highest error, deceptive jammer having the lowest error and constant and reactive causing similar errors. Moreover, by separating the errors between the topology types, we can see how the effectiveness of each jammer type changes based on the topology type. By looking at this, in a real-life situation, we have an idea of what type of jammer is more likely to be used, with respect to the topology, which ultimately helps us prepare for an attack or mitigate its effects.

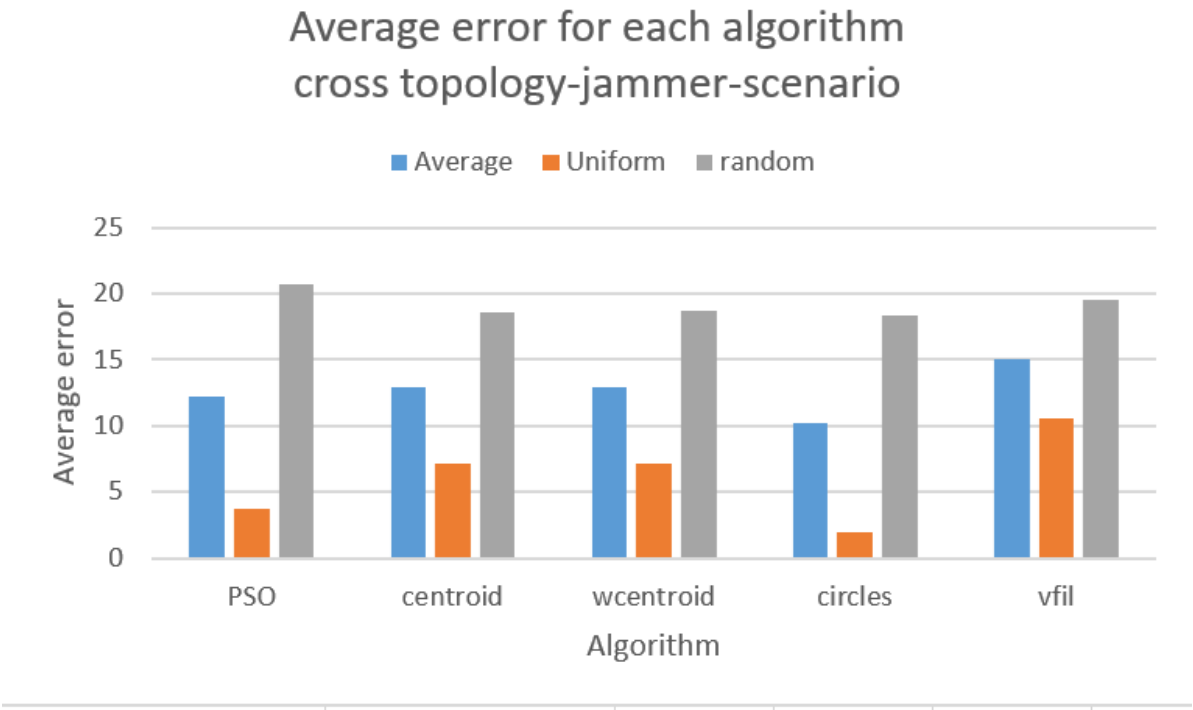


Figure 5.4.1.3

Figure 5.4.1.3 depicts the performance of each algorithm when all the results from all experiments are combined. This includes all jammer locations, all jammer types, and all topologies. By looking at this graph, we can tell that PSO and DCL are the most reliable as they have the lowest errors across the board. PSO is preferred over DCL since it always produces an estimation, whereas DCL sometimes provides no estimation at all. If we have a relatively smooth and uniform node distribution, DCL and PSO are almost identical in terms of localization accuracy. It is also worth noting that in a random topology, the five algorithms perform almost identically, leading to the conclusion that they are constrained by the quality of the readings.

5.4.2 Impact of Jammer Location on Overall Performance

By looking at the performance of the algorithms across the different jammer locations, our aim was to provide insights on how the jammer placement affects the localization accuracy.

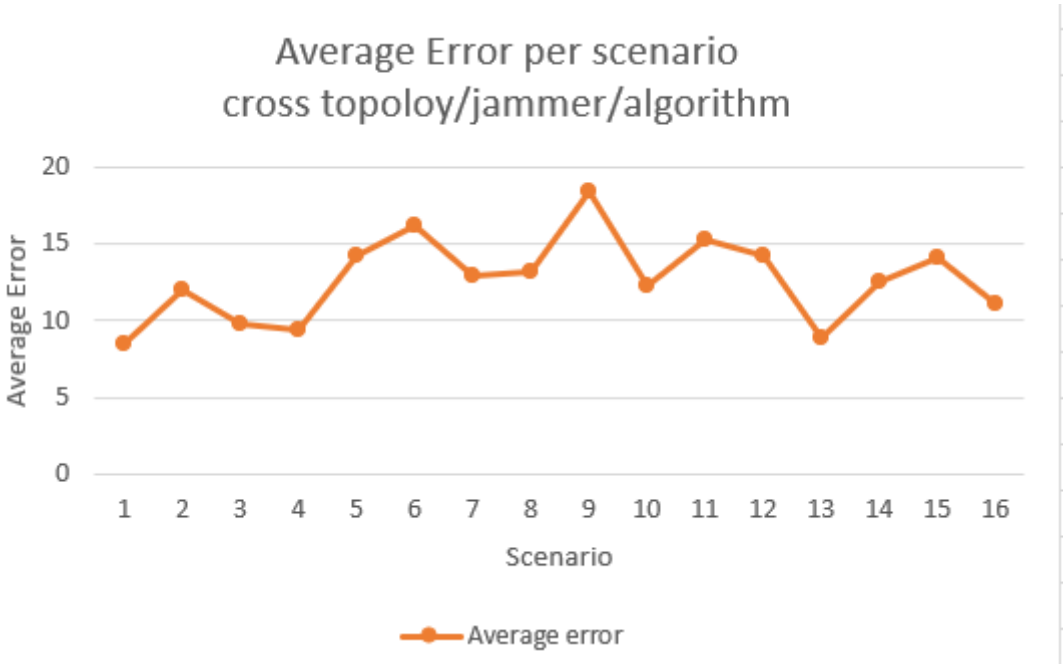


Figure 5.4.2.1

Figure 5.4.2.1 showcases the average error that each jammer location has caused, across both topologies, the four jammer types and all five algorithms. From the graph, we can see that there is virtually no difference in the localization error when the jammer changes positions. This comes to our surprise since we expected jammer locations closer to the sink to cause more error. However, this is not the case as jammer locations such as 6,7,10,11 do not have a localization error high enough to justify our speculation.

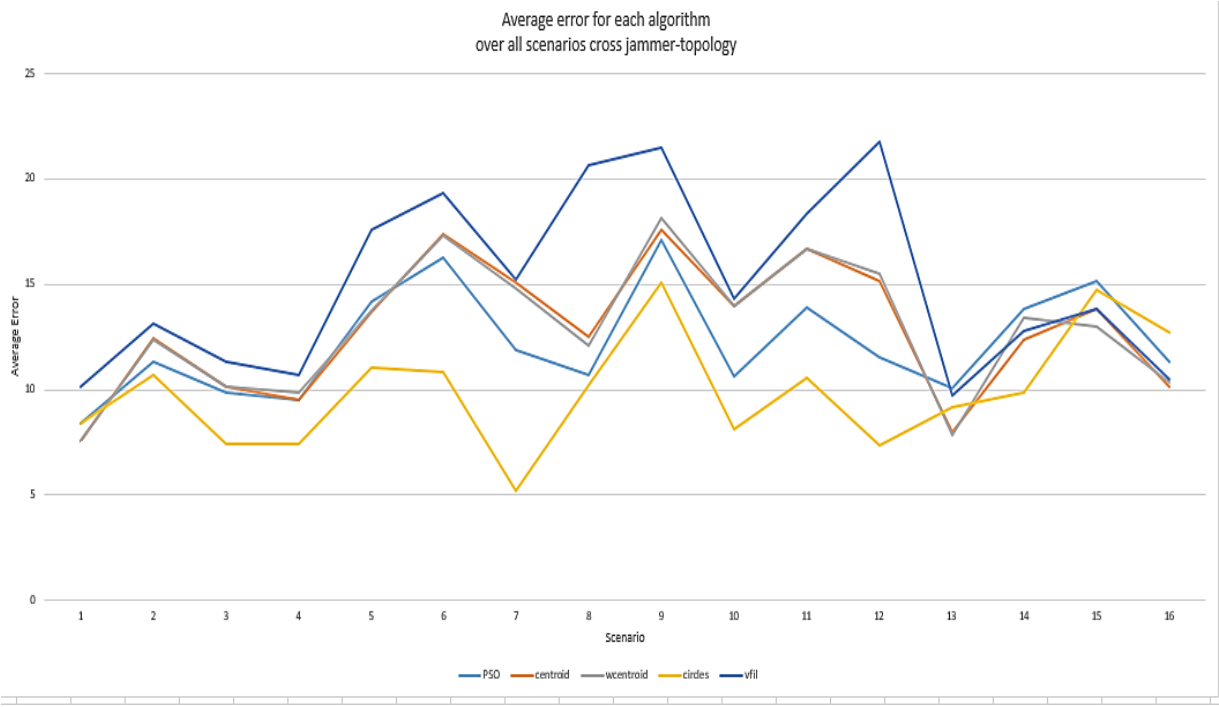


Figure 5.4.2.2

Figure 5.4.2.2 visualizes the performance of the algorithms by comparing them across each scenario. By comparing the algorithms like this, we can see that they follow the same trends when it comes to how they perform with respect to jammer placement. From what is shown above, we can confirm that there is no difference on how each algorithm performs when the location of the jammer changes. This means that no algorithm excels in a specific jammer location, so this information should not influence what algorithm one would choose to use.

5.4.3 CPU Execution Time Analysis

In the last subchapter of chapter 5, we present the analysis regarding the CPU execution times of the algorithms employed in our study. Our goal is to examine how the different parameters of the scenario, such as the topology, jammer type and jammer position influence the execution time for each algorithm.

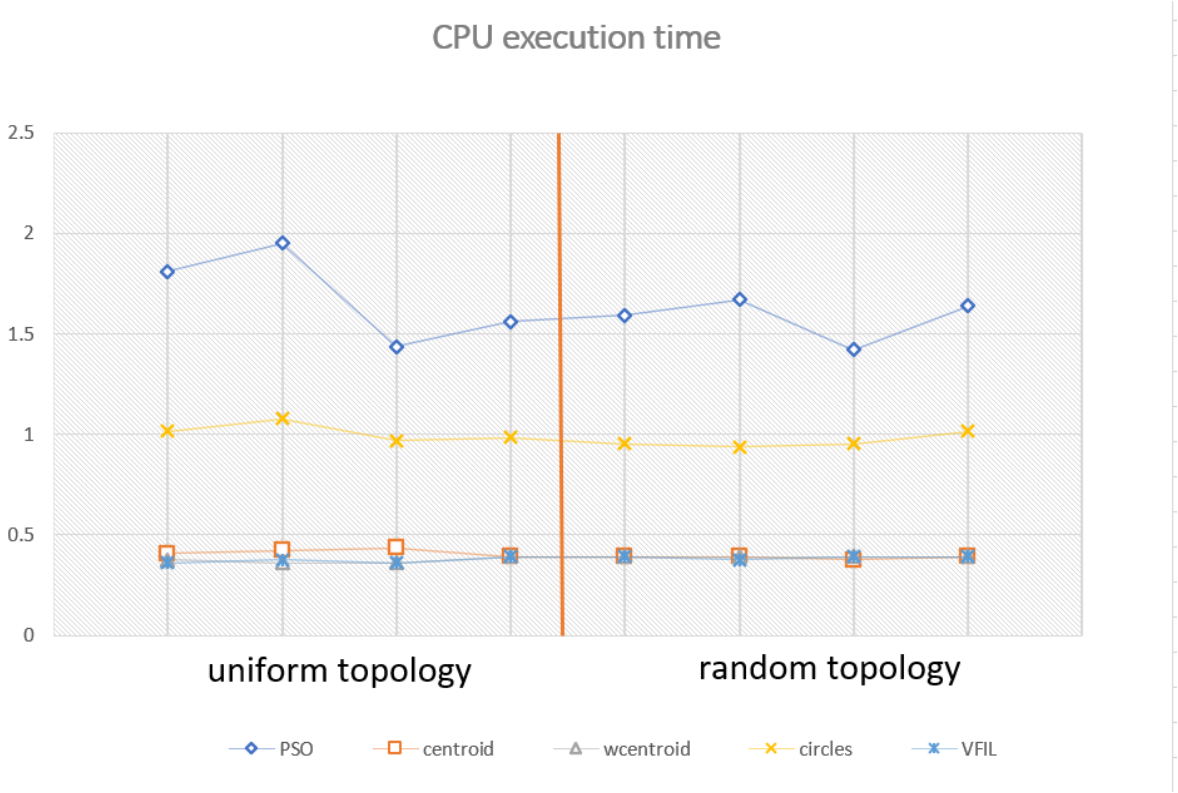


Figure 5.4.3.1 – CPU Execution Time

Figure 5.4.3.1 – CPU Execution Time, illustrates the CPU execution time of each algorithm across the two topology types and the four jammer types, as well as eight different jammer locations (2,4,6,8,10,12,14,16). As the graph suggests, there is no difference between the execution times across different topologies, jammer types and jammer locations. The graph is split in two parts, with the left part indicating the results from the uniformly distributed topology and the right part indicating the results from the randomly distributed topology. For example, the first four points for each algorithm, are simulations of the four different jammer types in the uniformly distributed topology, in the order of constant jammer, deceptive jammer, random jammer and reactive jammer. Each point represents a different combination of jammer type and jammer location.

From what is seen on the graph, each algorithm performs in the same way, regardless of the conditions of the experiment, which shows stability and trustworthiness. Contrary to the effect of jammer types and topology types in the localization accuracy, these do not influence the execution time of any of the five algorithms. This comes to our surprise, especially for the random jammer in the randomly distributed topology since it had caused the biggest localization error out of all scenarios in all algorithms.

Regarding PSO, the fluctuations in the execution times are caused by the inherent randomness that the algorithm incorporates. Since it utilizes a population of randomly generated particles and these particles are influenced by their personal and global best solutions, in combination with its stochastic nature, this leads to different values in the execution time. Updating each particle's position and velocity, also involves random factors, like the inertia weigh and the φ1,φ2 effective parameters. All these contribute to the variations in the execution times. Contrary to this, all the other algorithms are relying on predetermined calculations, which consequently leads to more consistent execution time. The fluctuations in CPU execution time that PSO exhibits, are not indicative of its performance, as we have seen that PSO is as consistent as the other algorithms and even better in some scenarios, especially in the uniformly distributed topology.

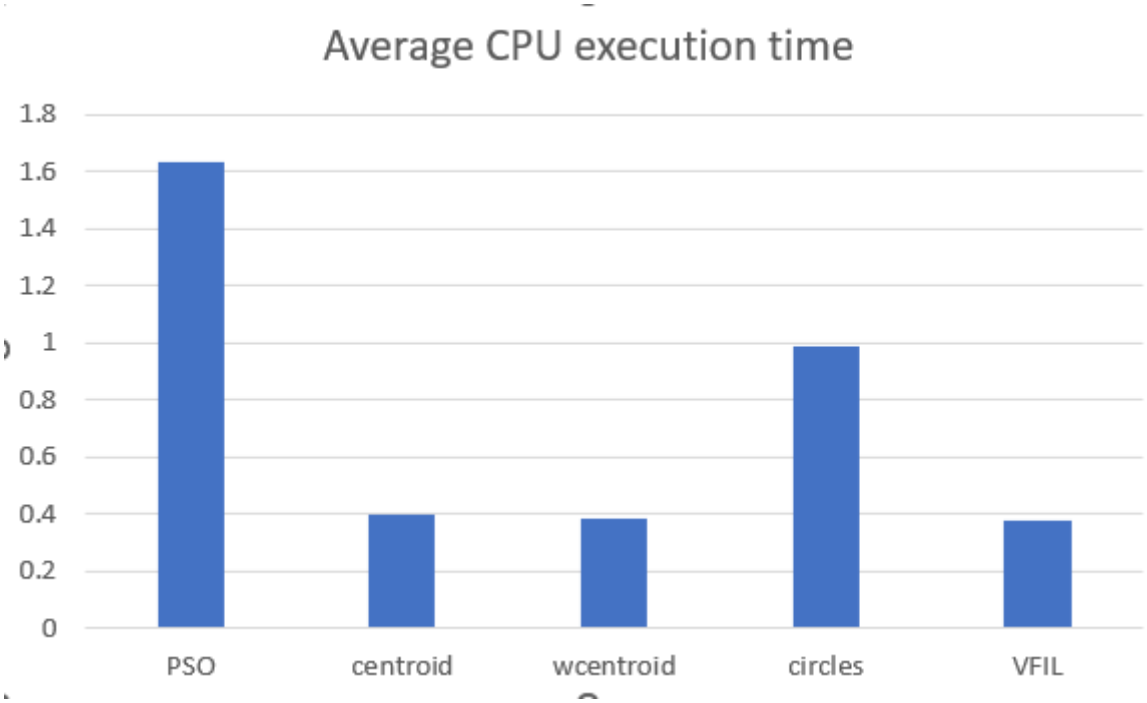


Figure 5.4.3.2 – Average CPU Execution Time

Figure 5.4.3.2 - Average CPU Execution Time, features the average CPU execution time of each algorithm, across all experiments. Since we have already clarified the effect of the scenario parameters is minimal on the execution time, we can proceed to compare the algorithms solely based on their mean execution times. It is evident that PSO takes far more time to complete (~1.6 seconds), which is something we expected since it is the most

complex out of the five algorithms reviewed in this work and it requires far more calculations.
DCL averages a CPU execution time of 1 second, which once again is expected as there is big function overhead from the functions calculating the convex hulls.
CL, WCL and DCL have the smallest average CPU execution time, near 0.4 seconds. Since they are the simplest algorithms, we had also expected them to have relatively low CPU execution times.

Chapter 6

Conclusion and Future Considerations

Arriving in this final chapter, we reflect on the extensive research and analysis conducted in this study, to present the most significant and meaningful insights mentioned throughout this paper. By studying the subject of jammer localization, throughout the study, we have aimed to aid towards developing effective techniques localizing jammers in wireless sensor networks. Moreover, we provide some valuable suggestions into the future work that could take place, to maximize the contribution in the jammer localization sector.

6.1 Important conclusions

From data collection and then algorithm development to simulation experiments and performance evaluations, we have managed to analyse the obtained result and reach important conclusions that could be of use to the ongoing battle against jamming attacks.

The most important conclusion of this study underlines the overall effectiveness of the selected algorithms in localizing the jammer within the network. Via conducting numerous experiments and simulations which feature different combinations of network parameters, such as the topology and the jammer type and position, we have managed to showcase that the selected algorithms are performing relatively good in localizing the jammer.

Secondly, we have managed to prove that the work was constrained by the quality of the provided data. Since we used different algorithms, that are implemented differently, work differently and approach the problem of jammer localization differently, we have conducted research that leads to clear conclusions. For example, we started with a very simplistic algorithm, CL that was used as our baseline example. WCL and VFIL are two amplified and enhanced versions of CL, that try to take advantage of other information that CL does not, to provide better results. From all the analysis presented in 5.2, we can see that in terms of localization error, these three algorithms perform similarly, across different scenarios. This indicates that, in the way WCL and VFIL are implemented cannot be of extra use and do not improve the results of CL, which leads to the conclusion that they are constrained by the quality of the data they were provided with. This can also be confirmed by the results of PSO, which performs at the same level as all the other algorithms when the data was of bad quality (e.g., the data collected from simulations where the topology had randomly distributed nodes). Having an algorithm which follows a completely different approach and that is much more computationally expensive perform in similar ways, is a sign that the capabilities of all algorithms are bound by the data.

We have also showcased that while DCL is ostensibly performing better than the other algorithms, we cannot rely on it for randomly distributed topologies, as a lot of times there wasn't enough information for the algorithm to reach a conclusion. If we also add the CPU execution times of DCL and PSO into the equation, we can say that it would be best to avoid using them completely in randomly distributed topologies.

Regarding the effects of different jammer types, different jammer locations and different topology types we can firstly say with certainty that there is a tremendous increase in the localization error for all algorithms when the topology is transitioned from uniform to random. No algorithm has managed to come up with particularly low localization errors in this type of topology. Moreover, jammers perform similarly with each other with respect to the jammer location, meaning that it doesn't really affect whether the jammer is placed in the middle, or in the corners of the grid. It is also important to point out it that both four jammer types are equally safe and dangerous in the uniform and random topologies respectively. The only significant difference was observed when the random jammer was used in the randomly distributed topology which showcased the largest localization error.

6.2 Suggestions

While this research features an extensive and comprehensive analysis and review of the performance of the algorithms, there are plenty of ways that it could be improved.
The selection of algorithms consisted of choosing approaches that are working on information similar with what we had in hand (e.g,. nodes record x,y,if-jammed). There is a huge collection of jammer localization algorithms from what we have seen in the literature review with other algorithms potentially being more suitable or better options that the ones we have. Moreover, the implementations for all algorithms except CL, follow some kind of variation which strafes away from what was initially proposed in the paper by the authors, meaning that these algorithms could be improved. For example, WCL could make use of a combined weight resulting from the fuzzy index plus the distance from the centroid, instead of only using the fuzzy index.
Moreover, PSO being an extremely complex algorithm can be improved in different points. There are many parameters that dramatically affect the performance of the algorithm, apart from the iteration and particle number, such as the effective parameters φ1, φ2, the social and cognitive weights, the inertia components and even the fitness function that evaluates the quality of each possible solution.

Lastly, since we have inferred that the overall performance is constrained by the quality of the data that the fuzzy index detection system has produced, perhaps a new and improved version of it could be reviewed. This improved version could potentially provide with better quality data. Combining this new improved data with the new and improved algorithms should result in better performances.

As we bring this research to a close, we need to acknowledge that the ever-growing field of WSN security is continuously evolving. Jamming attacks will become more and more prevalent and the anti-jamming measures will become even more demanding. We hope that the findings of this paper can contribute to the jammer localization field and inspire future researchers to join in this ongoing battle.

Bibliography

[1] Lu Tan and Neng Wang, "Future internet: The Internet of Things," 2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE), Chengdu, China, 2010, pp. V5-376-V5-380, doi: 10.1109/ICACTE.2010.5579543.

[2] Khalil, N., Abid, M.R., Benhaddou, D. and Gerndt, M., 2014, April. Wireless sensors networks for Internet of Things. In 2014 IEEE ninth international conference on Intelligent sensors, sensor networks and information processing (ISSNIP) (pp. 1-6). IEEE.

[3] Kocakulak, M. and Butun, I., 2017, January. An overview of Wireless Sensor Networks towards internet of things. In 2017 IEEE 7th annual computing and communication workshop and conference (CCWC) (pp. 1-6). Ieee.

[4] 3. Forum W.E. The Global Risks Report 2020. 2020. [(accessed on 21 December 2021)]. Available online: https://www.weforum.org/reports/the-global-risks-report-2020.html

[5] 9. Morgan S. Global Cybercrime Damages Predicted To Reach $6 Trillion Annually By 2021. 2018. [(accessed on 21 December 2021)]. Available online: https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/

[6] Y. Sun and X. Wang, "Jammer Localization in Wireless Sensor Networks," 2009 5th International Conference on Wireless Communications, Networking and Mobile Computing, Beijing, China, 2009, pp. 1-4, doi: 10.1109/WICOM.2009.5302614.

[7] Zadeh, L.A., 2023. Fuzzy logic. In Granular, Fuzzy, and Soft Computing (pp. 19-49). New York, NY: Springer US.

[8] H. H. R. Sherazi, R. Iqbal, F. Ahmad, Z. A. Khan and M. H. Chaudary, "DDoS Attack Detection: A Key Enabler for Sustainable Communication in Internet of Vehicles", Sustainable Computing: Informatics and Systems, vol. 23, pp. 13-20, 2019.

[9] J. E. Dickerson and J. A. Dickerson, "Fuzzy Network Profiling for Intrusion Detection", Fuzzy Information Processing Society 2000. NAFIPS. 19th International Conference of the North American, pp. 301-306, 2000.

[10] Savva, M., Ioannou, I. and Vassiliou, V., 2022, June. Fuzzy-Logic Based IDS for Detecting Jamming Attacks in Wireless Mesh IoT Networks. In 2022 20th Mediterranean Communication and Computer Networking Conference (MedComNet) (pp. 54-63). IEEE.

[11] Hongbo Liu, Wenyuan X, Yingying Chen and Zhenhua Liu, "Localizing jammers in wireless networks," 2009 IEEE International Conference on Pervasive Computing and Communications, Galveston, TX, 2009, pp. 1-6, doi: 10.1109/PERCOM.2009.4912878.

[12] J. Blumenthal, R. Grossmann, F. Golatowski and D. Timmermann, "Weighted Centroid Localization in Zigbee-based Sensor Networks," 2007 IEEE International Symposium on Intelligent Signal Processing, Alcala de Henares, Spain, 2007, pp. 1-6, doi: 10.1109/WISP.2007.4447528.

[13] Elhadi Shakshukia, Abdulrahman Abu Elkhailb, Ibrahim Nemerb, Mumin Adamb, Tarek Sheltamib, "Comparative Study on Range Free Localization Algorithms" The 10th International Conference on Ambient Systems, Networks and Technologies (ANT) April 29 - May 2, 2019, Leuven, Belgium https://doi.org/10.1016/j.procs.2019.04.068.
(https://www.sciencedirect.com/science/article/pii/S1877050919305307)

[14] T. Cheng, P. Li and S. Zhu, "An Algorithm for Jammer Localization in Wireless Sensor Networks," 2012 IEEE 26th International Conference on Advanced Information Networking and Applications, Fukuoka, Japan, 2012, pp. 724-731, doi: 10.1109/AINA.2012.11.

[15] Hongbo Liu, Wenyuan X, Yingying Chen and Zhenhua Liu, "Localizing jammers in wireless networks," 2009 IEEE International Conference on Pervasive Computing and Communications, Galveston, TX, 2009, pp. 1-6, doi: 10.1109/PERCOM.2009.4912878.

[16] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science, Nagoya, Japan, 1995, pp. 39-43, doi: 10.1109/MHS.1995.494215.

[17] Pang, L., Chen, X., Xue, Z., Khatoun, R. (2017). A Novel Range-Free Jammer Localization Solution in Wireless Network by Using PSO Algorithm. In: Zou, B., Han, Q., Sun, G., Jing, W., Peng, X., Lu, Z. (eds) Data Science. ICPCSEE 2017. Communications in Computer and Information Science, vol 728. Springer, Singapore. https://doi.org/10.1007/978-981-10-6388-6_17

[18] Xu, W., Trappe, W., Zhang, Y. and Wood, T., 2005, May. The feasibility of launching and detecting jamming attacks in wireless networks. In Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing (pp. 46-57).

Appendices

PSO Python Script:

```
# Particle Swarm Optimization Algorithm implementation      #
#                    ----------                              #
#                 Panayiotis Vasiliou                        #
#############################################################



import random
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import math
import time
np.set_printoptions(suppress=True)

st = time.time()
pst = time.process_time()

# Load the Excel file into a pandas DataFrame
# Extract the NodeID,x, y  in the node_coordinates array
nodes_df =
pd.read_excel("./Deceptive_random/random_nodes_coordinates.xlsx",sheet_name="
Sheet1")

node_coordinates = nodes_df[["Node","X", "Y"]].values


# Load the Excel file into a pandas DataFrame
# Extract the IMPORTANT colums C and H  in the jammed_info array
simulation_60sec_df =
pd.read_excel("./Reactive_random/reactive_random_ALL_values.xlsx",
sheet_name="fuzzy_(8)",usecols=[2,7])
jammer_x=60.00
jammer_y=-20.00
simulation_60sec_df = simulation_60sec_df.rename(columns={"Unnamed: 2":"Node"
, "Unnamed: 7": "Jammed"})

jammed_info = simulation_60sec_df.values


num_particles = 60
num_iterations=24
num_nodes=24
```

```
w = 0.65 # Inertia weight
c1 = 1.2 # Cognitive weight
c2 = 1.4 # Social weight

# This function takes the current position of a particle
# and a list of affected nodes (affected_nodes) and calculates the
# Euclidean distance between the particle and each affected node.
# The largest distance is returned as the fitness value, which
# represents the radius of the minimal covering circle.
# So fitness function is the minimal covering circle

#This is the PSO algorithm function.
# Particles are randomly generated withing the  search space , with a
# random velocity as well ( velocity in X coordinate and Y coordinate of the
search
# space respectively). Those particles influence one another on the way they
move , with
# each particle being a candidate solution to the problem. It is a
minimization problem
# as we are searching for the particle that has the smallest fitness value.
The fitness
# value is obtained by calculating the minimal covering circle around each
particle, with
# that circle containing each and every node in the network that currently
records that
# it is jammed. Hence , the smallest circle is the best solution. Particles
are affected
# by the values explained in notes , but they are mainly guided by their
personal best
# and by the global best alogn with some randomness.

# The algorithm begins by dividing the data into batches of 24 nodes.
Specifically in our tests
# we had a total of 14 batches. Each batch is basically the records of the
nodes for a specific
# period of time ( nodes record their jamming status). Each batch is
processed over 24 times before
# a final estimation is made regrding the precise location of the jammer.
# For each batch , we generate and initialize the particles. Then for 24
times the code goes over
# all the particles , one-by-one and calculates their minimal covering circle
by checking which
# nodes are jammed or not and adding the relevant ones to the circle. After a
particle is done
# with checking the 24 nodes of the current batch, we check for a personal
best and a global best
# and if necessary update those. Then we move to the next particle and do the
exact same thing.
```

```
# After it goes over all the particles 24 times, it moves to the next batch
and the same
# the set of particles.

# print ( "DATA LOADED ")
def calculate_fitness(jammed_info, node_coordinates, w, c1, c2):
    total_error=0
    # take each batch one-by-one ( agreed that there should be a result for
each batch )
    # reset the position, velocity, fitness etc
    for node_batch_index in range(14):
        jam_count=False
        particle_position = np.zeros((num_particles, 2))
        particle_velocity = np.zeros((num_particles, 2))
        particle_current_fitness = np.zeros(num_particles)
        particle_previous_fitness = np.zeros(num_particles)
        particle_personal_best_position = np.zeros((num_particles,
2))
        particle_personal_best_fitness = np.full(num_particles,
np.inf)
        global_best_position=np.zeros((1,2))
        global_best_fitness = float('inf')

        # regenerate the positions and velocities for the new batch
        for j in range(num_particles):
             particle_position[j] = np.random.uniform(0,100,2)
             particle_velocity[j] = np.random.uniform(0,100,2)
        max_distance = float('-inf')

        # pick the next batch
        node_batch = jammed_info[node_batch_index*num_nodes :
(node_batch_index+1)*num_nodes]

        #  start the processing for the batch ,( total of # iterations)
        for i in range(num_iterations):

            max_distance = float('-inf')

            # start picking up particles, one-by-one
            for particle_index in range(len(particle_position)):
                max_distance = float('-inf')
                current_position = particle_position[particle_index]
                current_velocity = particle_velocity[particle_index]
                # for each particle, go over all the nodes in the current
batch
                for node in node_batch:
                    # check for end of file
                    if node[0] > 26 or node[0] < 0:
```

```
                        break
                    # skip nodes that are NOT jammed
                    if node[1] != 1:
                        continue
                    else:
                        jam_count=True
                    # store the nodeID
                    node_id = node[0]
                    # link the current node being processed with it's
coordinates ( x,y in another excel file )
                    node_coords =
node_coordinates[np.where(node_coordinates[:, 0] == node_id)][0, 1:]
                    x_distance = current_position[0] - node_coords[0]
                    y_distance = current_position[1] - node_coords[1]
                    # euclidean distance between particle and node
                    distance = math.sqrt(x_distance**2 + y_distance**2)
                    # minimal covering circle, eg the fitness function is the
biggest distance ( furthest node)
                    if distance > max_distance:
                        max_distance = distance
                        particle_current_fitness[particle_index]=max_distance
                # do the necessary checks for personal and global optimum.
                # minimization problem , as we need the smallest circle
possible, therefore smallest value wins
                # update when
necessary
                if
particle_current_fitness[particle_index]<particle_personal_best_fitness[parti
cle_index]:
                    particle_previous_fitness[[particle_index]]=particle_pers
onal_best_fitness[particle_index]
                    particle_personal_best_fitness[particle_index]=particle_c
urrent_fitness[particle_index]
                    particle_personal_best_position[particle_index]=particle_
position[particle_index]
                if
particle_personal_best_fitness[particle_index]<global_best_fitness and
particle_personal_best_fitness[particle_index]!=0.0:
                    global_best_fitness=particle_personal_best_fitness[partic
le_index]
                    global_best_position[0][0] =
particle_position[particle_index][0]
                    global_best_position[0][1] =
particle_position[particle_index][1]
                # update particle velocity and position after iterating over
all nodes in the batch
```

```
                particle_velocity[particle_index] =
update_velocity(particle_index, particle_position, particle_velocity, w, c1,
c2, particle_personal_best_position, global_best_position)
                particle_position[particle_index] =
update_position(particle_index, particle_position, particle_velocity)
        # also add the error to the sum so that avg error can be calculated
        # avg error is important, as it tells us how good the algorithm is on
the specific scenartio ( jammer location )

        # print("batch : ", node_batch_index+1, "has calculated that the
position is : ",  (global_best_position))
        error=math.sqrt((jammer_x - global_best_position[0][0])**2 +
(jammer_y - global_best_position[0][1])**2)
        # mse = ((jammer_x - global_best_position[0][0])**2 + (jammer_y -
global_best_position[0][1])**2)/2
        # rmse = math.sqrt(((jammer_x - global_best_position[0][0])**2 +
(jammer_y - global_best_position[0][1])**2)/2)
        if(jam_count):
            print("batch : ", node_batch_index+1, " euclidean distance error
:  ", error)
            # print(error)

        else:
            print("batch : ", node_batch_index+1, " SKIPPED ")









    return 1

# function for the update of a particle's velocity.
def update_velocity(particle_index, particle_position, particle_velocity, w,
c1, c2, personal_best, global_best):
    r1 = random.uniform(0, 1)
    r2 = random.uniform(0, 1)

    velocity = particle_velocity[particle_index]
    velocity[0] = w * velocity[0] + c1 * r1 *
(personal_best[particle_index][0] - particle_position[particle_index][0]) +
c2 * r2 * (global_best[0][0] - particle_position[particle_index][0])
    velocity[1] = w * velocity[1] + c1 * r1 *
(personal_best[particle_index][1] - particle_position[particle_index][1]) +
c2 * r2 * (global_best[0][1] - particle_position[particle_index][1])
    return velocity
```

```
# function for updating a particle's postion
def update_position(particle_index, particle_position, particle_velocity):
    x = particle_position[particle_index, 0] +
particle_velocity[particle_index, 0]
    y = particle_position[particle_index, 1] +
particle_velocity[particle_index, 1]
    return np.array([x, y])
    # return particle_position


# calculate average error , while skipping first batch
average_error=(calculate_fitness(jammed_info,node_coordinates,w,c1,c2))/14

pet = time.process_time()

pres = pet - pst
print('CPU Execution time:', pres, 'seconds')
```

Centroid Localization Python Script :

```
import sys
import math
import time
import pandas as pd
pst = time.process_time()


df = pd.read_excel("./Reactive_random/reactive_random_ALL_values.xlsx",
sheet_name="fuzzy_(8)")
jammer_x=60.00
jammer_y=-20.00
array = df.values.tolist()


nodes_df =
pd.read_excel("./Constant_random/random_thesi.xlsx",sheet_name="Sheet1",
header=None)
node_batch_index=1
node_coordinates = nodes_df.values.tolist()

length = len(array)
count = 0  # counter to find how many 1 we have
```

```
arrayForX = 0
jam_count=0
arrayForY=0
sumX = 0
sumY=0
for i,j  in zip(array,node_coordinates):

    if((i[4])!=1.0 and (i[4]!=0.0)):
        continue
    count = count + 1

    if (i[4]) == 1.0:
        jam_count = jam_count + 1


    arrayForX = (i[4]) * (j[0])
    arrayForY = (i[4]) * (j[1])
    sumX = sumX + arrayForX
    sumY = sumY + arrayForY
    if count == 24:  # Once 24 nodes have been processed, calculate the
centroid
        if (jam_count==0):
            print ( " BATCH : " , node_batch_index, " skipped ")
            count = 0
            arrayForX = 0
            sumX = 0
            arrayForY = 0
            sumY = 0
            jam_count=0
            node_batch_index +=1
            continue;

        Xestimate = sumX / jam_count
        Yestimate = sumY / jam_count
        error=math.sqrt((jammer_x - Xestimate)**2 + (jammer_y -
Yestimate)**2)
        mse = ((jammer_x - Xestimate)**2 + (jammer_y - Yestimate)**2)/2
        rmse = math.sqrt(((jammer_x - Xestimate)**2 + (jammer_y -
Yestimate)**2)/2)
        count = 0
        arrayForX = 0
        sumX = 0
        arrayForY = 0
        sumY = 0
        jam_count=0
        # print("batch : ", node_batch_index, " has calculated :  ",
Xestimate , " ", Yestimate)
```

```
        print("batch : ", node_batch_index, " euclidean distance error :  ",
error)
        # print(error)

        # print("batch : ", node_batch_index, " mse distance error :  ", mse)
        # print("batch : ", node_batch_index, " rmse distance error :  ",
rmse)
        node_batch_index +=1

pet = time.process_time()


pres = pet - pst

print('CPU Execution time:', pres, 'seconds')
```

Weighted Centroid Localization Algorithm Python Script:

```
import sys
import math
import time
import pandas as pd
pst = time.process_time()


df = pd.read_excel("./Reactive_random/reactive_random_ALL_values.xlsx",
sheet_name="fuzzy_(4)")
jammer_x=60.00
jammer_y=-20.00
array = df.values.tolist()


nodes_df =
pd.read_excel("./Constant_random/random_thesi.xlsx",sheet_name="Sheet1",
header=None)
node_batch_index=1
node_coordinates = nodes_df.values.tolist()

length= len(array)
start_time = time.time()
count = 0 #counter to find how many 1 we have
weights=0
arrayForX = 0
jam_count=0
arrayForY=0
sumX = 0
```

```
sumY=0
for i,j  in zip(array,node_coordinates):

    if((i[4])!=1.0 and (i[4]!=0.0)):
        continue
    count = count +1
    if float(i[3]) > 0.575 :
        jam_count = jam_count + 1
        weights = weights +float(i[3])
        arrayForX= (i[3]) * (j[0])
        sumX = sumX + arrayForX
        arrayForY= (i[3]) *(j[1])
        sumY = sumY + arrayForY
    if count == 24:  # Once 24 nodes have been processed, calculate the
centroid
        if (jam_count==0):
            print("SKIPPED BATCH : ", node_batch_index )
            count = 0
            arrayForX = 0
            sumX = 0
            arrayForY = 0
            sumY = 0
            jam_count=0
            weights=0
            node_batch_index +=1
            continue;

        Xestimate = sumX / weights
        Yestimate = sumY / weights
        error=math.sqrt((jammer_x - Xestimate)**2 + (jammer_y -
Yestimate)**2)
        mse = ((jammer_x - Xestimate)**2 + (jammer_y - Yestimate)**2)/2
        rmse = math.sqrt(((jammer_x - Xestimate)**2 + (jammer_y -
Yestimate)**2)/2)
        count = 0
        arrayForX = 0
        sumX = 0
        arrayForY = 0
        sumY = 0
        jam_count=0
        weights=0
        # print("batch : ", node_batch_index, " has calculated :  ",
Xestimate , " ", Yestimate)
        # print("batch : ", node_batch_index, " euclidean distance error
:  ", error)
        print(error)

        # print("batch : ", node_batch_index, " mse distance error :  ", mse)
```

```
        # print("batch : ", node_batch_index, " rmse distance error :   ",
rmse)
        error=0
        node_batch_index +=1

pet = time.process_time()


pres = pet - pst

print('CPU Execution time:', pres, 'seconds')
```

Double Circles Localization Algorithm Python Script:

```
# double cicles algorithm

# input positions of jammed nodes and boundary nodes
# jammed nodes are the ones that have 1 in dataset.
# A node is considered to be a boundary node if it lost
# some of neighbors while it can communicate with part of unaffected nodes.
# output: estimated position

import math
import pandas as pd
import matplotlib.pyplot as plt
from scipy.spatial import ConvexHull, convex_hull_plot_2d, distance
from helper import Point, welzl
# import distance as d
import time
import gc

pst = time.process_time()

def calc_minimum_bounding_circles(boundaryhpoints, jammedhpoints):
    bhpoints = []
    for points in boundaryhpoints:
        bhpoints.append(Point(points[0], points[1]))

    jhpoints = []
    for points in jammedhpoints:
        jhpoints.append(Point(points[0], points[1]))

    # calculate circles of each

    mbc_bounding = welzl(bhpoints)
    mbc_jammed = welzl(jhpoints)
```

```
    return (mbc_bounding, mbc_jammed)



def calculate_jammer_position(filename, debug):
    start = time.time()
    data = pd.read_excel(filename,sheet_name="fuzzy_(4)")
    data.iloc[1]

    actualx = data['actual_X'].iloc[0]
    actualy = data['actual_y'].iloc[0]

    # get the number of nodes per batch
    nodes_per_batch = 24

    # get the total number of batches
    num_batches = 14
    # process each batch individually
    for batch_num in range(num_batches):
        boundary = None
        jammed = None
        boundary_hull = None
        jammed_hull = None
        mbc_bounding = None
        mbc_jammed = None
        # get the start and end indices of the current batch
        start_index = batch_num * nodes_per_batch
        end_index = (batch_num + 1) * nodes_per_batch

        # get the data for the current batch
        batch_data = data.iloc[start_index:end_index]



        # fuzzy column is un needed
        batch_data = batch_data.copy() # make a copy of the DataFrame slice

        batch_data.drop(columns=['Fuzzy'], inplace=True)

        # drop the unneeded columns
        batch_data.drop(columns=['actual_X'], inplace=True)
        batch_data.drop(columns=['actual_y'], inplace=True)



        # get unbounded nodes, the ones that have 0 in affected
```

```
        boundary = batch_data.drop(batch_data[(batch_data['affected'] ==
1)].index, axis=0)

        # remove affected column
        boundary.drop(columns=['affected'], inplace=True)

        # reset dataframe index
        boundary.reset_index(drop=True, inplace=True)
        boundary = boundary.to_numpy()

        # get jammed nodes, the ones that have 1 in affected
        jammed = batch_data.drop(batch_data[(batch_data['affected'] ==
0)].index, axis=0)

        # remove affected column
        jammed.drop(columns=['affected'], inplace=True)

        # reset dataframe index
        jammed.reset_index(drop=True, inplace=True)
        # convert to numpy array

        jammed = jammed.to_numpy()

        # calculate 2 convex hulls
        # one with the jammed nodes
        # one with the boundary nodes


        if ( len(boundary)<= 0 or len(jammed)<=0):
                    # reset relevant variables for the next batch
            print("SKIPPED BATCH : ", batch_num+1)
            del boundary, jammed, boundary_hull, jammed_hull, mbc_bounding,
mbc_jammed
            gc.collect()
            continue;

    # Check if there are enough boundary and jammed nodes
        if len(boundary) < 3 or len(jammed) < 3:
            print("SKIPPED BATCH : ", batch_num+1 , " NOT ENOUGH NODES TO
MAKE A HULL")
            del boundary, jammed, boundary_hull, jammed_hull, mbc_bounding,
mbc_jammed
            gc.collect()
            continue;

        if len(boundary) > 0:
            boundary_hull = ConvexHull(boundary)
        if len(jammed) > 0:
```

```
            jammed_hull = ConvexHull(jammed)




        if debug:
            print('Points:\n', boundary_hull.points)
            if (len (jammed>0)):
                print('Points:\n', jammed_hull.points)

        # calculate the minimum bounding circles




        mbc_bounding, mbc_jammed = calc_minimum_bounding_circles(
            boundary_hull.points, jammed_hull.points)

        # calculate the positon of the jammer
        x = (mbc_jammed.C.X-mbc_bounding.C.X)
        y = (mbc_jammed.C.Y-mbc_bounding.C.Y)
        # print(f'Batch {batch_num+1} - Predicted position: ', x, y)

        distance_error = math.sqrt((actualx-x)**2+(actualy-y)**2)
        print(f'Batch {batch_num+1} - Distance error: ', distance_error)
        # print(distance_error)

        # reset relevant variables for the next batch
        del boundary, jammed, boundary_hull, jammed_hull, mbc_bounding,
mbc_jammed
        gc.collect()

    stop = time.time()
    time_taken = stop-start







    return distance_error,time_taken


def main():
            error=0
```

```
            timeF=0
            filename = 'concat_values.csv'
            error, timeF =
calculate_jammer_position('./Reactive_random/reactive_random_concat_values.xl
sx', False)
            pet = time.process_time()


            pres = pet - pst

            print('CPU Execution time:', pres, 'seconds')



if __name__ == '__main__':
    main()
```

Virtual Force Iterative Localization Algorithm Python Script:

```
import sys
import time

import numpy as np
import math
import pandas as pd
pst = time.process_time()


def distance(x1, y1, x2, y2):
    one = (x1 - x2) ** 2
    two = (y1 - y2) ** 2
    return math.sqrt(one + two)

def main():
    start_time = time.time()

    df = pd.read_excel("./Reactive_random/reactive_random_ALL_values.xlsx",
sheet_name="fuzzy_(8)")
    jammer_x=60.00
    jammer_y=-20.00
    array = df.values.tolist()
```

```
    nodes_df =
pd.read_excel("./Constant_random/random_thesi.xlsx",sheet_name="Sheet1",
header=None)

    node_batch=0
    coordinate_batch=0


    node_batch_index=0
    coordinate_batch_index=0
    node_coordinates = nodes_df.values.tolist()

    length = len(array)
    start_time = time.time()
    count = 0  # counter to find how many 1 we have
    jam_count=0
    coordinate_batches = [node_coordinates[i:i+24] for i in range(0,
len(node_coordinates), 24)]
    node_batches = [array[i:i+24] for i in range(0, len(array), 24)]

    for node_batch, coordinate_batch in zip
(node_batches,coordinate_batches):
        Max = []
        maxX = []
        maxY = []
        sumX=0
        sumY=0
        arrayForX=0
        arrayForY=0
        count=0
        jam_count=0
        Xestimate=0
        Yestimate=0
        Xestimate2=0
        Yestimate2=0
        max_d = 0
        max_x = 0
        max_y=0
        temp=0
        F_Pull = []
        F_Push = []
        pullX=0
        pullY=0
        pushX=0
        pushY=0
        sum_sqX =0
        rootX = 0
        sum_sqY =0
```

```
        rootY =0
        DistanceJamRange = 0
        sum_x_pull = 0
        sum_y_pull = 0
        sum_x_push = 0
        sum_y_push = 0
        SumJoint=0



        for i, j in zip(node_batch,coordinate_batch):

                if(int(i[4])!=1.0 and int(i[4]!=0.0)):
                        continue
                count=count+1
                if int(i[4]) == 1.0:
                    jam_count = jam_count + 1


                arrayForX = (i[4]) * (j[0])
                arrayForY = (i[4]) * (j[1])
                sumX = sumX + arrayForX
                sumY = sumY + arrayForY


        if (jam_count==0):
            print ( "node_batch ID : ", node_batch_index +1,
"SKIPPED")
            count = 0
            arrayForX = 0
            sumX = 0
            arrayForY = 0
            sumY = 0
            jam_count=0
            node_batch_index+=1
            coordinate_batch_index+=1

            continue;
        Xestimate = float(sumX / jam_count)


        Yestimate = float(sumY / jam_count)
```

```
        for i,j in zip(node_batch,coordinate_batch):

                if int(i[4]) == 1.0:
                    temp = distance(Xestimate, Yestimate, float(j[0]),
float(j[1]))
                    if temp > max_d:
                        max_d = temp
                        max_x = j[0]
                        max_y=j[1]

        maxX = float(max_x)
        maxY = float(max_y)

            # Estimated area

        Xestimate2 = float(Xestimate)
        Yestimate2 = float(Yestimate)
            # Pull, Push Formula
        sumPullX = 0
        sumPullY = 0
        sumPushX = 0
        sumPushY = 0

        for i, j in zip(node_batch,coordinate_batch):

                if float(j[0]) == Xestimate2 and float(j[1]) == Yestimate2:
                    pass
                else:
                    if int(i[4]) == 1.0:

                        pullX = (float(j[0]) - Xestimate2) / math.sqrt(
                            math.pow(float(j[0]) - Xestimate2, 2) +
math.pow(float(j[1]) - Yestimate2, 2))
                        pullY = (float(j[1]) - Yestimate2) / math.sqrt(
                            math.pow(float(j[0]) - Xestimate2, 2) +
math.pow(float(j[1]) - Yestimate2, 2))

                        F_Pull.append({'x': pullX, 'y': pullY})

                    else:

                        pushX = (Xestimate2 - float(j[0])) / math.sqrt(
                            math.pow(Xestimate2 - float(j[0]), 2) +
math.pow(Yestimate2 - float(j[1]), 2))
                        pushY = (Yestimate2 - float(j[1])) / math.sqrt(
                            math.pow(Xestimate2 - float(j[0]), 2) +
math.pow(Yestimate2 - float(j[1]), 2))
```

```
                        F_Push.append({'x': pushX, 'y': pushY})

        sum_sqX = np.sum(np.square(pushX - pullX))
        rootX = np.sqrt(sum_sqX)

        sum_sqY = np.sum(np.square(pushY - pullY))
        rootY = np.sqrt(sum_sqY)


        DistanceJamRange = rootX + rootY



            # Joint Formula
        for index, z in enumerate(F_Push):

                sum_x_push = sum_x_push + z['x']
                sum_y_push = sum_y_push + z['y']

        for index, z in enumerate(F_Pull):

                sum_x_pull = sum_x_pull + z['x']
                sum_y_pull = sum_y_pull + z['y']

        SumJoint = (sum_x_push + sum_x_pull) / abs(sum_y_push + sum_y_pull)
        SumJoint = (sum_x_push + sum_x_pull) / abs(sum_y_push + sum_y_pull)

        Xestimate2 = Xestimate2 + SumJoint * DistanceJamRange
        Yestimate2 = Yestimate2 + SumJoint * DistanceJamRange

        # print("New estimated jammer's position:", Xestimate2, Yestimate2)

        timet=time.time() - start_time

        error=math.sqrt((jammer_x - Xestimate2)**2 + (jammer_y -
Yestimate2)**2)
        print("Batch num : ", node_batch_index+1,"The error rate is:", error)
        # print(error)

        coordinate_batch_index+=1
        node_batch_index+=1



    pet = time.process_time()
```

```
    pres = pet - pst

    print('CPU Execution time:', pres, 'seconds')


if __name__ == '__main__':
    main()
```