ADAPTING CANONICAL PARTICLE SWARM OPTIMIZATION TO A SWARM OF KILOBOTS IN EVENT LOCATION TASKS

A thesis presented to the faculty of the Graduate School of
Western Carolina University in partial fulfillment of the
requirements for the degree of Master of Science in Technology.

By

Matthew George Stender

Director: Dr. Yanjun Yan
Assistant Professor
School of Engineering and Technology

Committee Members: Dr. H. Bora Karayaka, School of Engineering and Technology
Dr. Peter Tay, School of Engineering and Technology

April 2018

This work is dedicated to my mother

for raising me to be an intelligent gentleman

And especially to my father

for always believing in me.

ACKNOWLEDGEMENTS

I would first like to acknowledge my adviser Dr. Yanjun Yan for her patience, kindness and most of all her assistance in completing this thesis. I would like to thank Dr. H. Bora Karayaka and Dr. Peter Tay for serving on my thesis committee. To all those mentioned, and additionally Dr. Robert Adams, I would like to extend thanks for helping my papers get published. I would also like to thank Dr. Erin McNelis for referring me to the program.

Table of Contents

List of Tables

List of Figures

ABSTRACT

ADAPTING CANONICAL PARTICLE SWARM OPTIMIZATION TO A SWARM OF KILOBOTS IN EVENT LOCATION TASKS

Matthew George Stender, M.S.T.

Western Carolina University (April 2018)

Director: Dr. Yanjun Yan

In nature, there are many species who are tiny and simple as individuals, but are very organized and effective as a group, for foraging, defense, and other tasks. This phenomenon has inspired the development of swarm robotics, which has been applied from simulating nano-particle based medication administering to controlling hundreds of UAVs in formation for ceremonial display and/or surveillance. This dissertation aims to explore the idea of swarm intelligence, in a search and rescue simulation scenario, to establish a test-bed and to make the idea practical for the control of a swarm of robots. In order to do this efficiently, the robots will have some swarm intelligence method to govern their behavior. One such swarm intelligence method is Particle Swarm Optimization (PSO), originally developed by Kennedy and Eberhart in 1995 as a way of modelling natural swarm behavior. There are several popular options of swarm robots from research groups and we chose Kilobot designed by Harvard Self-organizing system research group, the most cost efficient option. The team at Western Carolina University has improved/updated the design while building a few dozen of these robots. Therefore, the experimenting agent is set to be Kilobot, and the practical constraints of Kilobots, such as communication range, movement mechanisms, are all taken into account when adapting the idea of PSO for swarm robotic control. To address the issues of limited communication range of Kilobot and measurement noises, we first proposed the Neighborhood PSO (NPSO) algorithm and examined it in the Matlab simulation environment. Three benchmark functions are used to simulate the measurements on the interest level at each location: when there is an emergency event like a fire, the fitness value at that location will be higher than that in its neighboring region (for calculation simplicity, though, the fitness is assumed to be the

smaller the better, and the global minima is with a fitness value of 0.) Monte Carlo simulations are carried out given the random nature of the algorithms, and the results are reported in convergence speed, accuracy and consistency. Once NPSO was established as comparable to PSO in Matlab simulations, we adopted the NPSO idea into a more realistic Kilobot simulation environment, Kilombo, in Linux. Kilombo has incorporated many practical aspects of Kilobots, such as its physical size, its moving and turning speed, and its communication channel. The code developed in Kilombo should be portable to Kilobots. In Kilombo, the Kilobots' movements can be sped up saving simulation time. The Kilobots are given the tasks of locating the spot with the best fitness, simulating the search of an event of interest. The fitness values are provided by the call-back functions when Kilobots inquire such values, simulating their measurements. We then propose a new motion mechanism, called pseudo-vector motion (PVM), inspired by our NPSO algorithm to control the swarm. We have proposed another three PVM-based algorithms subsequently to address the issues that are observed in the Kilombo simulation experiments.

CHAPTER 1: Introduction

1.1 Background and Motivation

As controllers are becoming smaller and cheaper, micro-robot swarms are more accessible for conducting research on the behavior of multi-agent systems. These swarms may be used for many different purposes, such as shape aggregation or event tracking. A more practical application of these types of swarms could be applied in situations where it is too dangerous for human intervention. Many of these tasks, such as locating a fire, detecting a radiation leak, or disposing of an explosive, are being done by manned robots. Using a swarm of autonomous robots will free people from the hazardous or repetitive tasks. When applying these robot swarms in such an application, it would be ideal for these robots to interact as efficiently as possible, in order to accomplish the tasks as quickly as possible.

Swarm intelligence research has provided us many cooperative examples in which multiple simple agents working together can achieve things too large or complex for any single agent to achieve on its own. Advancements in computer optimization techniques provide a broad basis of swarm intelligence techniques to draw from. For the purpose of this research, a computationally simple, distributable algorithm is desirable. One such optimization technique is Particle Swarm Optimization (PSO), originally proposed in 1995 by Kennedy and Eberhart [2]. PSO was conceived as a way of modelling bird-flocking or fish-schooling, and it has received much attention in recent years for its ability to find an optimize solution quickly with a low computational load.

When applying PSO to a micro-robot swarm some challenges arise based on the robot's capabilities. The Kilobot [3], designed by the Self-Organizing Systems Research Group at Harvard University is an accessible agent on which to conduct this research. Instead of using a fitness function as in a simulation, the Kilobots have onboard sensors to search for some optimum value. The robots are unable to immediately reverse direction and their communication range is limited.

1.2 Objectives

The objectives of this study are as follows

1. Adapt PSO to the physical constraints of the Kilobots.
2. Examine whether the adapted PSO is comparably effective as canonical PSO.
3. Develop event-locating algorithms for Kilobots.
4. Incorporate the adapted PSO algorithm into Kilobot control.

1.3 Significance of the Study

Adapting PSO into the physical constraints of the robots serves multiple purposes. Not only does it provide an algorithm more likely to be integrated into a swarm system, it also shows the robustness of PSO as an optimization technique. The adaptation of PSO into swarm robotic control has been validated in this thesis under limited communication range, measurement noise, communication packet loss, and the restricted movements of a robot.

CHAPTER 2: Background

2.1 Particle Swarm Optimization

Particle Swarm Optimization (PSO) was developed in 1995 by James Kennedy and Russell Eberhart [2] as an optimizer to train a back-prorogation neural network by mimicking the social behavior of bird flocking or fish schooling. PSO is applicable in a high-dimensional solution space and is efficient to find a solution quickly [4]. A PSO consists of a swarm of particles initialized in a search space whose locations represent solutions. The particles then move through the space searching for the position of best fitness. A particle's fitness is a measure of how good the current solution (represented by the particle's current location in the solution space) is. The fitness function is available to all the particles for them to evaluate their position instantly. A particle keeps track of the position that has given its best fitness, and all the particles are able to communicate with each other to find the position that gives the global best.

In PSO, a population of particles is used. Initially each particle is assigned a uniformly distributed random location and a uniformly distributed random velocity. The particles then move through the solution space driven by three factors (will be explained soon), but with randomization [5, 6]. The randomness in the velocity and position update ensures the exploration ability of the particles to avoid pre-mature convergence and stagnation.

The PSO algorithm is summarized in the pseudo code, Algorithm 1. $\mathbf{x}_i(t)$ is the current position of the ith particle at time instance t. $\mathbf{v}_i(t)$ is its current velocity. Positions in the search space represent solutions to a fitness function. $\mathbf{p}_i(t)$ is the best position x_i has seen, i.e. the particle's personal best known position (pBest). $\mathbf{g}(t)$ is the best known position amongst all the particles, i.e. the global best known position (gBest).

Initially when $t = 0$, $\mathbf{p}_i(t)$ is set as $\mathbf{p}_i(0) = \mathbf{x}_i(0)$ for all particles. $\mathbf{g}(0)$ is initialized as the best among $\mathbf{p}_i(0)$. Once iterations begin, the positions and velocities of all the particles are

Algorithm 1 PSO algorithm

Require: Randomly initialize particle position and velocity: $\mathbf{x}_i(0)$ and $\mathbf{v}_i(0)$

```
while terminating condition is not met do
    for i = 1 to number of particles, N do
        Evaluate the fitness f(x_i(t))
        Update p_i(t) and g(t)
        Update the particle position using (2.1) and (2.2)
    end for
end while
return g(t)
```

updated by (2.1) and (2.2) recursively.

$$\begin{aligned}\mathbf{v}_i(t+1) =& \omega \mathbf{v}_i(t) \\ &+ C_1\varphi_1(\mathbf{p}_i(t) - \mathbf{x}_i(t)) \\ &+ C_2\varphi_2(\mathbf{g}(t) - \mathbf{x}_i(t))\end{aligned} \tag{2.1}$$

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{2.2}$$

where ω, C_1, and C_2 are weights. $0 \leq \varphi_1, \varphi_2 \leq 2$ are two uniformly distributed random numbers.

In the first term of (2.1), the previous step's velocity is taken into account, although weighted, representing the inertia or momentum of the particle. In the second term, the coefficient C_1 weights the motion of the particle towards pBest, representing the particle's self-cognizance. In the third term, the coefficient C_2 weights the motion of the particle towards gBest, representing the social influence.

During iterations, each particle in the swarm is aware of its position. The particle first checks if its current position is better than pBest. If it is, pBest is updated by the current solution. Then the particle checks if its current position is better than gBest, and updates gBest when needed.

The particles keep updating their positions and velocities as they search for the optimum, until the termination condition is met, such as a predetermined number of iterations, an accuracy threshold for the algorithm to achieve, or an upper limit of CPU usage.

PSO has been applied in many applications, such as neural network training to obtain the synapses' weights in a much quicker way than back-propagation [4, 7]. It has also been used

for optimization of multi-modal functions, such as optimization of reactive power and voltage control [4]. It has been applied to estimate parameters in dynamic systems [8] or to control robotic swarms [9, 10].

PSO is efficient for robotic learning in swarm robotics, due to its relatively low computational cost, which is critical in a real-time system. Since the only info that the particles have to share is the global best position and do not need a central controller to process all the information, it makes PSO ideal for a distributed system [9, 10]. Several researchers have applied PSO for robotic search applications. Hereford et al [9] used mitEBots for light-tracking and Di Mario et al [10] used Khepera III robots to analyze the randomness of robotic performance.

PSO has been used to solve many different problems and has a performance comparable to other stochastic-based optimization techniques, such as Genetic Algorithms, Differential Evolution, Simulated Annealing, etc. [5]. PSO is similar to Genetic Algorithms in that they both initialize a population with random values, each representing a solution; and then the solutions evolve in different ways, as PSO evaluates the velocities for each potential solution to move to a new solution, while Genetic Algorithms do chromosome mutations, cross-overs, and selection to generate a new set of population [11].

There are several advantages of PSO over other aforementioned search algorithms. PSO is computationally simple and efficient, allowing for fast processing speeds. Each particle only needs to know its local information and the information from its neighbors to perform its computations, so there is minimal amount of communication between particles. The results from all particles in the population are not required for the individual particle's computations, except the global best that the population has seen, so no extensive calculations are needed at the central controller. PSO also has very few parameters to adjust compared to other search algorithms. These advantages make the PSO conceptually simple and easily implementable. PSO has also been shown to converge quickly, many of the times quicker than Genetic Algorithms for some problems with relatively low-dimensional search spaces [8].

PSO, though it tends to be quicker than other algorithms for small search spaces, deterio-

rates as the size and dimension of the search space increases. In the basic PSO, particles tend to fly towards the global best position. It is this social interaction that allows for the quick discovery of good solutions, but also lends to the PSO prematurely converging at a local optimum [5, 7]. Premature convergence could be alleviated by adapting the weights of the three factors affecting the search at different stages of the search. Many studies have been done on parameter tuning in order to attempt to improve the convergence of PSO [4, 5, 7–9, 11–15]. PSO is useful in search applications as it needs little assumptions about the problem in order to optimize it [12].

There are other variants of PSO that attempt to fine tune parameters in order to improve convergence. Many of these variants involve changing the initial distribution of the solutions in the search space. Using Gaussian, Lognormal and Exponential distributions has shown improvement on the performance of the convergence of the PSO [5]. Other variants change the way the velocity weight parameters change. Instead of changing the velocity weights linearly, changing them exponentially can improve the clustering of data sets [7]. Another study has shown that randomizing the acceleration constants under a Gaussian distribution can also improve the performance when in conjunction with a Gaussian solution initialization [12]. Further adaptations use quasi-random sequences to initialize the swarm instead of using a uniform distribution. These sequences are shown to improve upon the convergence of PSO versus randomly distributed initialization, because they cover the search space more evenly [16]. Other adaptations will add elements of other algorithms into the mix, such as Gaussian Mutation [16] or Crossover [17].

Much of the research on PSO models the stability of the algorithm's behavior in the presence of Gaussian noise [10, 18–21]. This allows simulations to compensate for the inaccuracies of real-world measurements. In addition, alterations to the search space are made, such as rotating the entire space, in order to provide a model for changing environments [18]. These studies show that PSO can be disturbed by fitness noise as it offsets the early learning stages of the algorithm. It has been shown that parameter tuning alone does not improve upon the convergence when under the effect of noise [10, 19], and that PSO may not be effective for optimization problems in the presence of a large noise factor [20]. PSO is more sensitive to noise than Genetic Algorithms [21]. In this thesis, multiple noise levels have been examined.

2.2 Kilobot

In the research of swarm robotics, one of the limiting factors in experimentation is the cost of individual robots. Since robot costs are high and operation of these robots can be complex, it is difficult to build a truly collective swarm to model robotic behavior. The Harvard Kilobot (as shown in Figure 1) is an attempt at making affordable and computationally-capable robot swarms [3]. The robots are quoted at being build-able for $14 each, but that is when buying parts for an exceptionally large of swarm and the cost of using a pick-and-place machine is not included, although such a machine is nearly a must. We have found the costs to be close to $70 per robot in our efforts to build the Kilobots. While $70 is much higher than $14, $70 is still reasonable for our research purposes.

Figure 1: A Kilobot

Oh, Shirazi and Jin have used the Kilobots for tracking a light source [1]. The Kilobots then follow the light source and herd around it. The Kilobots are herding around the light source while maintaining swarm coherence. In another word, the Kilobots keep other robots within a certain distance of themselves as they progress through this task. The authors developed the heuristic control algorithm for their Kilobots, and have shown its effectiveness in both numerical simulation and physical experimentation.

Rubenstein, Cornejo and Nagpal have used the Kilobots for self-assembling complex forms [3].

Some Kilobots are placed as a "seed" and other Kilobots then locate the seed and form a specified shape around it based on a binary bitmap preloaded by the user. The control algorithm they developed for this purpose is multi-faceted and very complex, using gradient formation, localization and edge-following techniques in tandem. They have performed successful self-assembly experiments for shapes like a star or a wrench using 1024 robots. Having shown the capabilities of a Kilobot swarm, this research supports that the Kilobot is a good benchmark robot to further the analysis of PSO governing a particle swarm.

2.3 Simulators

There are many restrictions to test algorithms on a physical Kilobot swarm. The Kilobots require calibration and regularly need to be recharged. The recalibration and charging will decrease the time allotted for physical experimentation. A simulator would be a quick test bed for experimenting the Kilobot control algorithm, and three simulators were available: V-REP, kbsim, and Kilombo.

V-REP is a robotic simulation suite with integrated development environment [22]. The developers at K-Team have provided both a model and several scenes for Kilobots in the V-REP simulator. The simulator is powerful, but also daunting with its IDE. Its scripts are written in LUA, which requires interfacing with C.

kbsim is a simulation environment for Kilobot written in Python using the pygame library [23]. It provides both a simulator and a pattern designer. This software requires a complete port from Python to C in order to deploy the algorithm on a physical swarm of Kilobots. Figure 2 shows a sample output of a swarm configuration generated by kbsim [1,23]. In this example a light source is represented by a black dot with a large circle showing its effective range and the robots are represented by the colored dots with smaller circles showing their communication ranges. The red dots are herding towards the light source, the blue dot is moving straight and the green dots are turning around to maintain swarm coherence. kbsim is used to simulate some of the most important behaviors of the Kilobots, such as sensing, communication and movement [1]. However, it

lacks sensing noise and disturbance, motor control and true physical interaction.

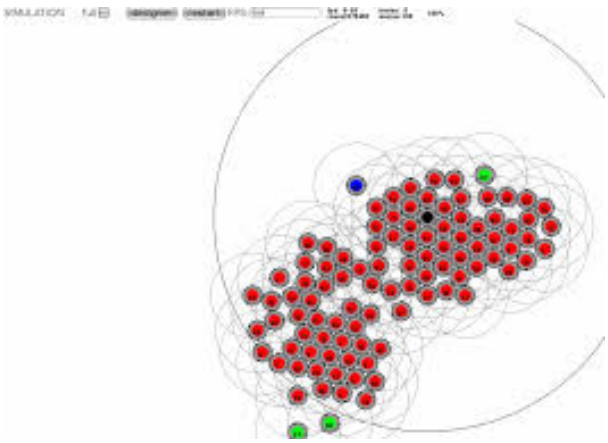


Figure 2: An example run of kbsim. [1]

Kilombo was developed by Jansson et al [24] to address the challenges in the first two simulators. Namely, it intends to be sufficiently accurate, efficient, and quick to deploy code between simulator and Kilobot. It comes with many premade examples, such as two types of gradient lighting, orbiting and following-the-leader. The simulator is available on GitHub under the MIT license. More details about the Kilombo simulator are in [24]. We decided to use Kilombo, and one of the simulation screen-shot is in Figure 3. The small circle with an arrow inside represents a Kilobot, where the arrow indicates the moving direction. The big circle around a Kilobot is its communication range. The green trace is its latest past trajectory, which is refreshed as the simulation goes on.

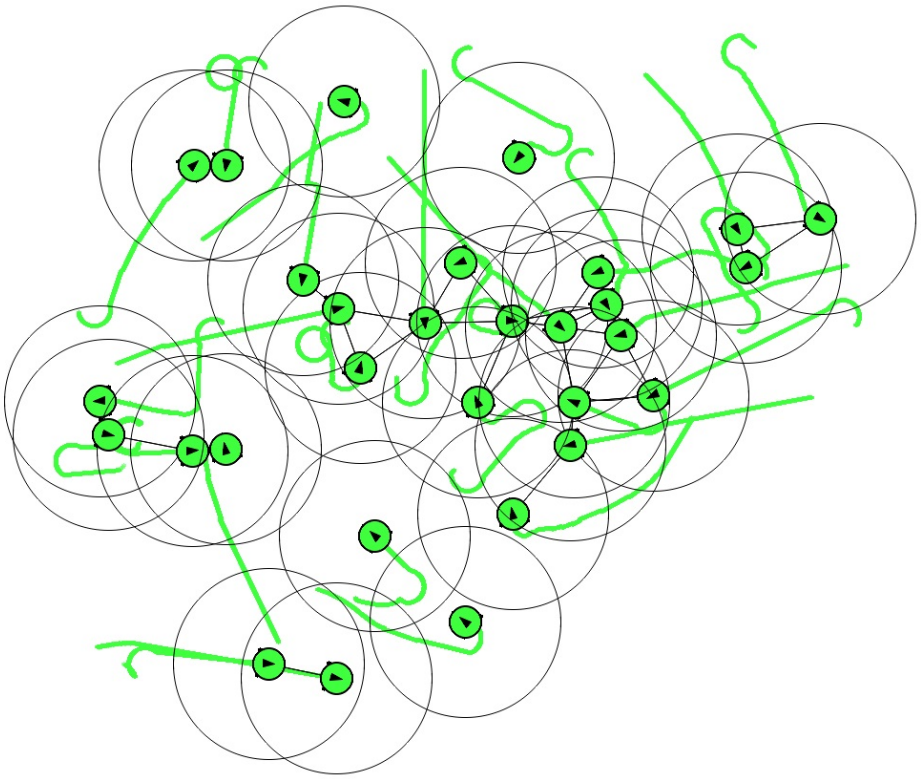


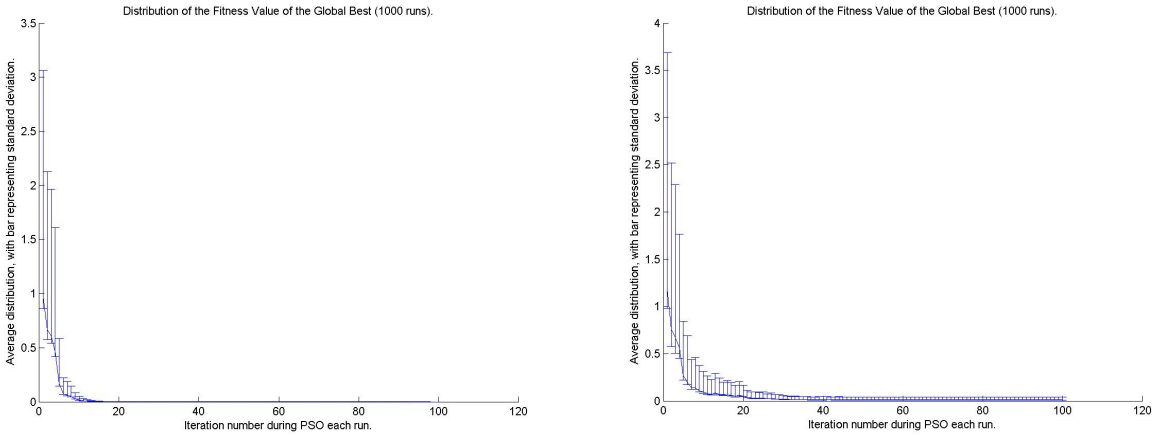
Figure 3: A Kilombo simulation screen-shot

CHAPTER 3: Methodology

3.1 Particle Swarm Test

Since its invention, the PSO algorithm has been applied on a multitude of optimization benchmark functions, such as the Spherical function and the Rastrigin function to find the minimum of those benchmark functions. A similar set of tests has been carried out in this thesis, but under noises and other physical constraints of a robot. The progression of the global best is recorded during iterations. A Monte-Carlo simulation is performed on each benchmark function to see how the gBest converges over time, especially under the influence of noise. The metrics to measure the results include the number of iterations needed to converge, and the accuracy and consistency of the final result from the algorithm.

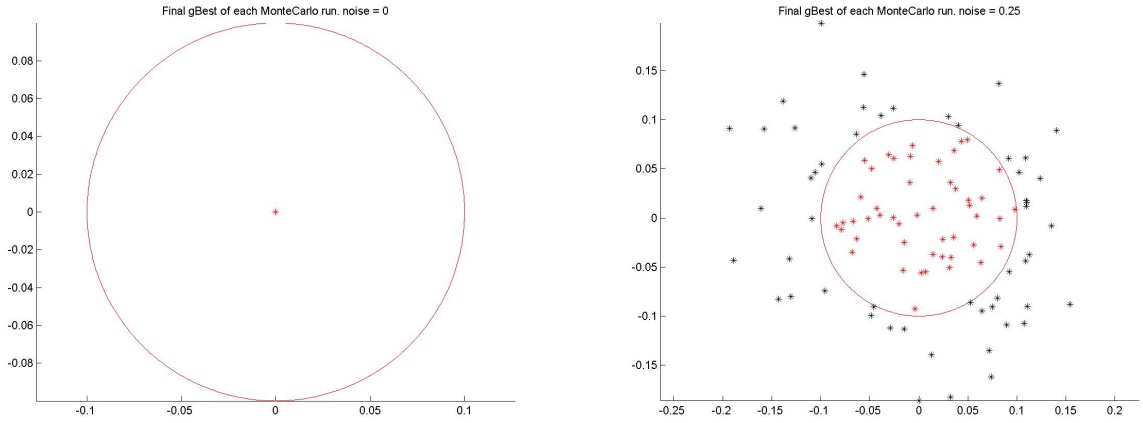
An example of the convergence of the PSO algorithm is shown in Figure 4, with error bars representing the deviation of the gBest at each iteration of Monte-Carlo (MC) runs. We carried out 100 Monte Carlo runs for each algorithm.



(a) when noise is small (b) when noise is big

Figure 4: Error Bars

For lower noise levels (in Figure 4 (a)), the fitness value range converges to a single point; namely, all the MC runs are finding the global optima. For higher noise levels (in Figure 4 (b)), the fitness value range does not converge to a single point; namely, some of the MC runs land in the neighborhood of the global minimum but not exactly at it. The accuracy of the final gBest from each Monte-Carlo run is shown in Figure 5. In the physical system, Kilobots have a diameter of about 33 mm, and finding the global optima means that the global minimum is covered by any part of a Kilbot, and hence the center location of the Kilobot may not necessarily be the exact point of the global minimum. Therefore, the gBest can be deemed successful if it falls within a distance of the global minimum, as represented by the circle in Figure 5 (b). This tolerance radius is determined based on the size of Kilobot, to compensate for the physical size limitation.



(a) without noise (b) with noise

Figure 5: Accuracy Figures

To control a swarm of Kilobots, an overhead controller (OHC) conveniently transmits infrared commands to the entire swarm. Once the common commands are transmitted, the OHC is no longer used, and the swarm of robots move autonomously based on their own measurements and decisions. Each Kilobot is equipped with 32KB ISP flash memory, 1024B EEPROM (Electri-

cally Erasable Programmable Read-Only Memory), and visible-light and infra-red sensors. Each Kilobot communicates with other Kilobots within its communication range.

Given the measurement noise and table-lookup accuracy to estimate the distance of another Kilobot from measured light intensity, Kilobots do not have an accurate and precise fitness function evaluation as in the canonical PSO. Multiple noise levels (standard deviations of the Gaussian noise) are experimented in the fitness function evaluation to account for variations due to noise, and Monte-Carlo simulations are carried out to assess the performance of the PSO algorithm under various noise levels.

Another challenge in using PSO in swarm robotics is that unlike the particles in the canonical PSO that can access gBest in all the iterations instantly, the communication range of each robot is limited. A broadcasting scheme is too expensive to be feasible as it drains the battery. Therefore, practically each particle only communicates with the other particles within its neighborhood [25–28]. Monte-Carlo simulations are carried out to investigate the efficiency of using neighborhood best (nBest) instead of gBest. This adapted algorithm is called Neighborhood PSO (NPSO).

The NPSO algorithm begins in the same way as in the canonical PSO, but during iterations, each particle saves nBest instead of gBest. When a particle updates its position, it compares the fitness value of its current position to that of its nBest. It updates its nBest if this current position is better (smaller, in the fitness definitions in this thesis). It then compares its nBest with neighboring particle's nBest and updates the nBest of all the particles within this neighborhood, when necessary. The neighborhood will be defined as any particles within 70 mm as this is the Kilobot's specified communication range. Since each neighborhood is defined relative to a center particle, all the particles in the population take turns to be such a center particle of a neighborhood. The last particle in the update has the most information aggregated from all the previous particles' updates, and the first particle has the least information. Therefore, an arbitrary index (the i value of each particle x_i) is assigned to each particle at the initial step, with i between 1 and N. Then, when the time step index t is odd, the center particles are exhausted in the order of their ascending indices (i

goes up from 1 to N); when t is even, the exhaustion is in a descending order (i goes down from N to 1). This ordering scheme ensures a thorough update of all the particles.

No matter what kind of application is studied, the four following scenarios are examined and compared.

1. The first simulation scenario is the canonical PSO with perfect fitness knowledge and gBest, as a baseline performance for comparison.

2. The second scenario is the canonical PSO with noisy fitness evaluation, to study the effect of noisy fitness evaluation alone.

3. The third scenario is NPSO with perfect fitness knowledge, to study the effect of using nBest alone (instead of gBest).

4. The fourth scenario is NPSO with noisy fitness evaluation, to study the combined effect of both noisy fitness and using nBest instead of gBest.

All the noise is Gaussian, and the noise level is the standard deviation, which is experimented at various values. The noise levels in simulation are chosen based on the steepness of the fitness value surface, or the fitness value difference between global and local optimal points on the fitness value surface.

The NPSO validation experiments are carried out in Matlab simulations, and the particles in NPSO are still essentially mathematical massless points. The next step is to simulate the NPSO idea in a more realistic simulation environment where each Kilobot has a physical dimension and its moving and turning are governed by mechanisms.

3.2 Software Model Test

Kilombo simulator is adopted in this thesis to test and adapt the NPSO idea to control the Kilobots in a more realistic environment than in Matlab.

Every swarm needs a basic set of rules that each agent can follow independently. This set of rules should be tailored to the task the swarm is attempting to accomplish. In our case, we need the swarm to search an emergency event, such as fires, mimicked by a light source. The K-Team has a basic light finding algorithm based on the location of the ambient light sensor relative to forward motion. The Kilombo simulator calculates the light intensity based on a call-back function using each robot's x-y location as the truth, which is available only at the back end of the simulator and Kilobots have no knowledge of their locations. In another word, the measurement of light intensity is simulated by reading the output value of a built-in function based on the location of the Kilobot. By simply changing the built-in function, we can simulate various solution spaces and events, which is another convenience of using the Kilombo simulator.

The NPSO algorithm has been shown in Matlab simulations to be effective for robotic control [29], despite its infeasible features of the particles, and the ideas of NPSO are the inspirations for the control algorithms in this section. In the NPSO algorithm, there are three driving forces to guide the movement of a particle: (1) the momentum, or the particles tend to move in the same direction as its current direction; (2) cognitive awareness, or the knowledge on where the best place is based on each particle's past knowledge; (3) social influence, or the knowledge from neighbors on where the best place is given all the neighbors' past knowledge. The neighborhood is defined as a subset that are within the communication range of a particular particle. The three driving forces are weighted by random numbers to yield a combined guidance on where to move in the next step, to ensure both exploration and exploitation abilities of an agent.

A couple of constraints not present in the mathematical Matlab simulation must also be taken into account in the Kilombo simulator. First, particles of a physical dimension cannot overlap so that once a robot finds the target and occupies the place then other robots cannot take up the same place of the target. In addition, Kilobot measurements from the ambient light sensor are quantized into integers, simultaneously reducing the exactness of measurements and increasing the size of the area with the same measured light intensity level, although the actual light intensity could have a slight gradient in this area. Therefore, a new metric to declare success is used to include the congregation of Kilobots that are touching the Kilobot that occupies the best location.

Second, Kilobots do not have specific hardware for positioning, so they cannot move into a location based on coordinate as the coordinate information is not available. In the Matlab simulation, the amount of movement between steps is represented by the coordinate difference vector (called vector motion). The vector motion needs to be adapted in Kilombo and physical Kilobot experiments as the instant vector movement with arbitrary displacement and direction is not feasible. Kilobots also take time to rotate and cannot instantly turn to an arbitrary orientation without moving in an arc. A pseudo-continuous method to replace the vector motion from NPSO will address these issues.

3.2.1 Pseudo-Vector Motion (PVM)

The kilolib API provides us with a measurement of time for the Kilobots called kiloticks. There are about 31 kiloticks per second. Using kiloticks as the unit of time, if long enough time is allowed for the Kilobots to move forward or rotate by certain degrees before their next movement, the movement of a Kilobot will be adequately finished and similar to the vector-motion in Matlab simulation. Hence, this movement scheme is called pseudo-vector motion (PVM). We can also set a number of kiloticks a Kilobot has to spend at the same ambient light level before registering a success event to ensure that it is not just passing the success location by accident.

Every Kilobot runs a main loop function in the simulator. In our most basic version of this main loop, the Kilobots will not change their motion until certain number of kiloticks has passed. This amount of time for the bot to react unhindered allows us to simulate a rigid "vector motion" as in NPSO [29]. At the start of the simulation, an arbitrary direction to turn is chosen for all Kilobots. Unless a Kilobot turns completely around to reverse its turning direction, it will keep its current turning direction when a 30° turn is needed. Kilobots take measurements from its ambient light sensor continuously. Assuming a Kilobot has started a new step, it then uses the latest light intensity measurement as a comparison with the next light intensity measurement. This Kilobot will decide and record if the new measurement is better, the same, worse, or if it has found success. A simple state machine is as follows:

- If light intensity measurement is improving, move forward.
- If light intensity measurement is staying the same or getting worse, turn about 30° in its currently set direction.
- If light intensity has stayed the same long enough, register success and stop.

This rule is summarized in the first row of Table 3.1. PVM is the first stage of algorithm in Kilombo simulations. The next three rows in Table 3.1 are the movement rules for the next three PVM-based stages. These four stages also differ in the Kilobots' success-declaration rules, and such rules are summarized in Table 3.2. The details of each new stage will be discussed in the subsequent sections.

Table 3.1: Difference in PVM and Cognitive motions based on last measurement.

Algorithm	Better Measurement	Same Measurement	Worse Measurement
PVM	Move Forward	Move Forward	Turn 30°
CPVM S-CPVM CS-CPVM	Move Forward	Move Forward 3 Steps. Turn 30° 4th step.	Turn 30° 3 Steps. Turn 180° 4th step (back the opposite direction).

Table 3.2: Success declaration for different algorithms in the Kilombo simulator.

Algorithm	**Success Requirement**
PVM	in the **same measurement** for 300 **consecutive** kiloticks.
CPVM	in the **personal minimum** for 300 **consecutive** kiloticks.
S-CPVM	in the **neighborhood minimum** for 300 **consecutive** kiloticks, or find an already successful agent.
CS-CPVM	in the **neighborhood minimum** for a 300 **cumulative** kiloticks, or find an already successful agent.

Regardless of the step-time, each Kilobot will constantly transmit infra-red (IR) signals to the other bots to calculate the distance between two robots. If two bots get too close, the one with the lowest ID (assigned in calibration arbitrarily and stays the same in one experiment) will stop in an attempt to allow the other to move away. However, without directional and positioning hardware on the bots, sometimes this is sub-optimal when the stopped bot is in the path of the moving bot, because the moving bot will just push the stopped bot out of the way. This basic algorithm is the first one being tested, as a baseline to compare with later improvements. The stopping condition for this algorithm is not robust, and the bots tend to continue to search until the simulation stops. The expected behavior near the end of a successful search should be a swarm of agents around the best locations (determined by the built-in fitness function in Kilombo). This portion of the algorithm corresponds to the inertia in the NPSO algorithm, as the Kilobots tend to move along its previous trajectory.

3.2.2 Cognitive Pseudo-Vector Motion (CPVM)

The next portion of NPSO to be incorporated into swarm control is the personal cognizance factor. Each agent will remember how their personal measurement has been changing and what motion they were taking as it changed, beyond inertia. However, Kilobots lack a global positioning system, so the agents won't be able to remember where they made this measurement. While Kilobots lack the ability to perform vector calculations because of this, they can do additional comparison allowing agents to turn back towards a found minimum quicker. Kilobots are now able to change the amount of time between steps arbitrarily. This allows for turns at specified angles, such as a complete turn-around. In the simulator a Kilobot takes 256 kiloticks to make a 180 degree turn. Now, whenever a measurement is getting worse, the Kilobot will change its turn direction and then turn completely around. A resulting problem with this algorithm is that other agents do not notice the success of a single agent and have a tendency to push a successful agent out of position in order to locate the same event. In order to maintain the measurement found when registering success, this CPVM algorithm will only update a Kilobot's light intensity measurements if the Kilobot has not registered a success event yet, otherwise it will maintain its best-known location at

the time of success. This will be the second algorithm being tested.

Specifically, as summarized in Table 3.1 on CPVM, when a Kilobot senses the same measurement from last update, it will move forward. This can happen for three consecutive steps, but if the measurement is still the same at the 4th step, it will turn slightly to try to get out of the monotonous region. The cycle then continues. When the measurements get better, the Kilobot will move forward. When the measurements get worse, the Kilobot will turn to try to get out of it.

As shown in Table 3.2, CPVM differs from PVM in that the success declaration is based on staying in the best fitness that Kilobot has seen personally but not just staying in the same measurements long enough.

3.2.3 Social-Cognitive Pseudo-Vector Motion (S-CPVM)

The final portion governing NPSO to be emulated in the Kilobots is the social influence. The agents will now communicate between each other. The lack of a global communication between agents is why NPSO was tested and found to be acceptable. Allowing communication, agents pass their current measurements to other agents within their neighborhood, and are capable of remembering the ID and measurement of any other agent whose measurement is better than their own. This measurement is stored in a 16 bit integer, but the package sent between robots only holds an array of 8 bit integers. For this reason we shift the bits of the message in order to send it, and unpack and reconstitute these bits back into a measurement on the receiving end. These agents also pass whether they register success in their messages. This solves the problem of the previous algorithm when a Kilobot searching for success pushes an already successful one out of the way. Instead other bots that get close enough to a successful Kilobot now also stop their search and register success in response. This creates a star-like cluster of robots around any found minimums. At this point our loop function is becoming large enough that the pieces are modularized. This will be the third algorithm being tested.

S-CPVM shares the same movement rule as CPVM as shown in Table 3.1, but S-CPVM's success declaration rule differs from CPVM in that the Kilobot needs to find and stay in the best

fitness area within its neighboring cluster of Kilobots (see Table 3.2).

3.2.4 Cumulative Social-Cognitive Pseudo-Vector Motion (CS-CPVM)

As the complexity of locating an event increases due to multiple minimums, obstacles, or other agents, the Kilobots begin having problems locating a success event. To remedy this we make the stopping condition adaptive. After an arbitrary amount of time of searching, each agent will begin to accumulate the time it has spent to always measure the same as its best instead of resetting the timer whenever the Kilobot happens to get out of the best zone. This accumulated duration to be in the best is used to determine when the Kilobot stops movement. If the minimum changes for any reason, such as receiving a better measurement in a message, the running total is reset to 0 and the accumulation will start again if the Kilobot measures a new best during multiple steps.

3.2.5 Kilombo Simulator Setup

The simulator has many variables that are set in a json file that is loaded into the simulator upon each run. This file contains variables such as time for the simulator to run, the percentage of successful message packet delivery, how much distance noise exists in the simulation and so on. Regardless of what fitness function will be used to determine light intensity, all the simulations will use a population of 30 Kilobots and each algorithm will be run 100 times as in Monte-Carlo simulations. The simulation begins with all the bots randomly distributed in the search space, as in canonical NPSO. The parameters of the disturbances to the Kilobots' search task include both distance measurement noise levels and message success rate between robots. Each simulation will run for 500 seconds (15501 kiloticks) in order to give any straggling robots time to register a success upon either finding an event or spending enough time in a minimum. This is a reasonable amount of time to observe the behavior of all four algorithms, as well as adequate for the Kilobots to register a success event. The noise is assumed to be Gaussian. To help extract the simulation results data, Kilombo provides a json state file to save the states to.

CS-CPVM shares the same movement rule as S-CPVM and CPVM as shown in Table 3.1,

but CS-CPVM's success declaration rule differs from S-CPVM in that the time duration for the Kilobot to stay in the best area is accumulative, instead of being restarted, every time the Kilobot happens to wander away from the best area (see Table 3.2).

CHAPTER 4: Results

4.1 Mathematical Results Using Modified Particle Swarm Algorithm

4.1.1 Results Using Spherical Fitness Function

In a fire emergency, the sooner the rescue crew agents (at least one of them) get to the fire site, the better. This scenario is simulated by using a light source as the fire, and the Kilobots, as the rescue crew agents, randomly distributed within the region to move towards the event. The fitness function here is the distance between Kilobot and the light source, based on received light intensity. With a single event, the fitness function is a scaled spherical function with a single minimum in the search space, as defined in (4.1)

$$f(\mathbf{x}) = \sum_{j=1}^{d} x_j^2 \tag{4.1}$$

where $\mathbf{x}$ in $f(\mathbf{x})$ is the particle position, but its index i and time step t are omitted, as this fitness evaluation is common to all particles at all times. d is the dimension of the fitness function, which is set to be 2 in a 2D space, but it could be other integer values at higher dimensions. x_j is the jth coordinate of the particle position $\mathbf{x}$. These notations are similarly used in (4.2) and (4.3). In this thesis, the fitness functions are adapted from benchmark optimization functions, which are often used to evaluate and compare optimization algorithms [30]. The adaptation is aimed to scale the search space to match the white-board area that the Kilobots are moving on, and to scale the fitness values to be in comparable ranges of practical scenarios. All the coordinates in our simulations are in the unit of millimeters with identical search space.

In the simulation experiment, the swarm consists of 30 particles in a two-dimensional search space with a range between -556.9 and 556.9 millimeters in the x dimension and -444.5 and 444.5 millimeters in the y dimension, matching the search area of the white-board that Kilobots are moving on. After test runs for weight parameter selection, the weight for the inertia component is set to be 0.5, the weight for pBest component is set to be 2, and the weight for gBest component

is set to be 1. In each run, the particles are updated in 100 iterations. Then 100 Monte Carlo runs are carried out in each scenario.

Fig. 6 reports the simulation results by a quad-chart. In each quadrant, the performance of each scenario is reported in two plots: a fitness progression plot, and a histogram of the distance between gBest and the true minimum.

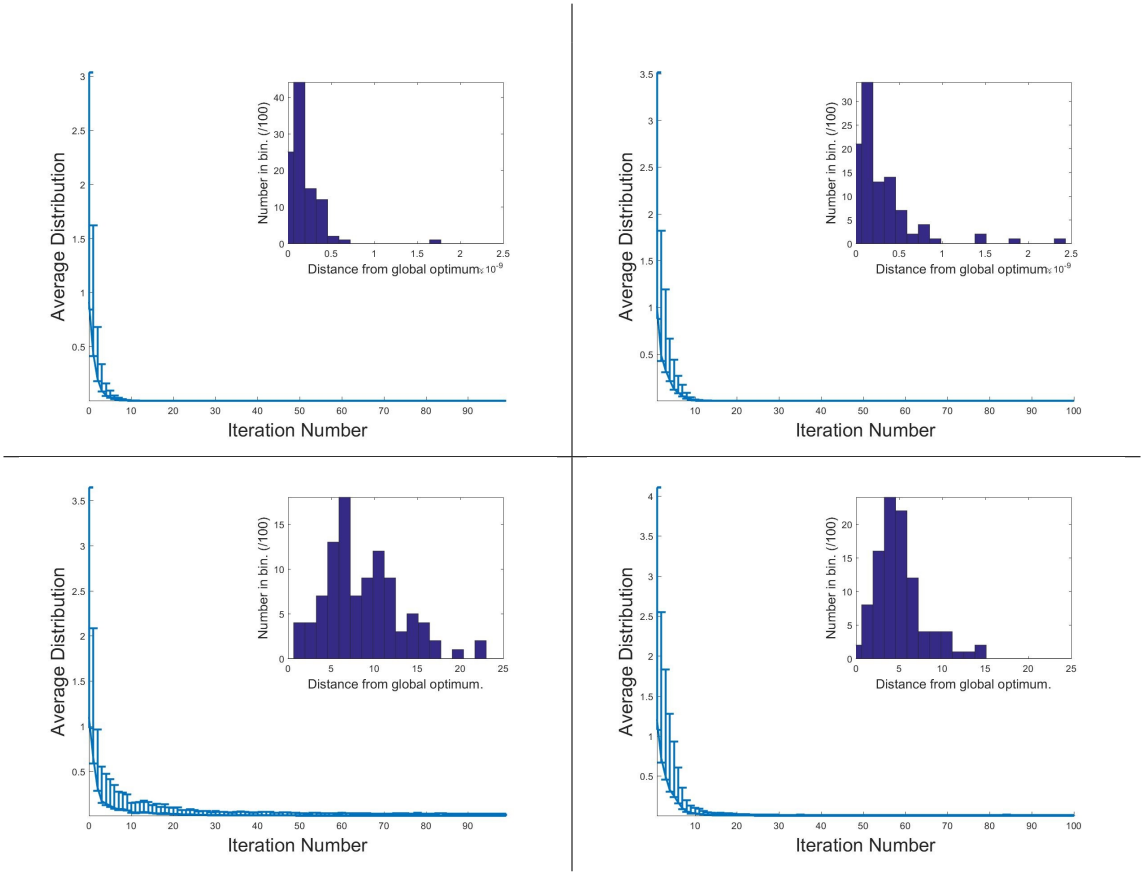


Figure 6: Results with a scaled spherical fitness function. The performance of each scenario is reported in a fitness progression plot, and a histogram of the distance between gBest and the true minimum. The left column uses PSO, and the right column uses NPSO. The top figure in each column is without noise, and the bottom figure is at a noise level of 0.25.

In the quadrants of Fig. 6, the left column uses PSO, and the right column uses NPSO. The first row is without noise, and the second row is at a noise level of 0.25. Namely, the top-left

quadrant is the first scenario described in Section 3.1, the bottom-left is the second, the top-right is the third, and the bottom-right is the fourth.

The large plot in each quadrant is the fitness progression plot, reporting variations in 100 Monte Carlo simulations. At each iteration step, the middle bar is the average value, the top bar is the 90 percentile, and the bottom bar is the 10 percentile. The quicker the fitness gets down to 0, the quicker the convergence. The thinner the spread of the variation at each iteration, the more robust the convergence.

The smaller plot in each quadrant is the histogram of the distance between the final best positions (gBest in PSO or nBest in NPSO) and the true optimum point. As one can see, without noise, the distance is in the order of 10^{-9}, or essentially 0 perfectly. But with noise at a level of 0.25, the final position of best position shows a deviation from the optimum.

It is worth-mentioning in this case that when there is noise, NPSO achieves a quicker convergence and a smaller distance than PSO. PSO and NPSO exploration involves randomness and when the measurement noise is globally present, the neighborhood information could help speed up the converge.

4.1.2 Results Using Rastrigin Fitness Functions

Spherical function models a single-event scenario, but when there are multiple events with one of them being the most severe and needing the most urgent attention, the searching of this global optimum, while local optima are present, becomes challenging. For example, when a fire starts in a pipeline system, the fire site is the global optimum with the highest temperature, but due to the pipe structure, the conjunctions might be overheated to be at a higher temperature than the connecting pipes and these conjunctions become local optima. Only if the fire (the global optimum) is located and extinguished, the local optima will disappear for good. The difference in fitness (such as temperature in the aforementioned example) between the global optimum and local optima may be small, making the identification of the global optimum even more challenging. The

Rastrigin function, as defined in (4.2), is a multi-modal function with many local minima.

$$f(\mathbf{x}) = 10d + \sum_{j=1}^{d} [x_j^2 - 10\cos(2\pi x_j)] \tag{4.2}$$

The surface and contour plots of a scaled 2D Rastrigin function are shown in Fig. 7 and Fig. 8 respectively. The large dot in the center of the contour plot shows where the global minimum is. The centers of other circular contour groups are local minima and maxima. The local minima are positioned in a checker-box fashion: the closest layer of local minima is 368 millimeters away from the global minima; The second layer of local minima is 520 millimeters away from the global minima.

Using the scaled Rastrigin fitness function, the same experimentation setup is used as that of the spherical function, except the noise level. The fitness values between adjacent layers of minima differ by 14.75 (The global minimum and the first layer of local minima differ in fitness values by 14.75; the local minima at adjacent layers also differ in fitness values by 14.75). If the error in noisy fitness evaluation is big enough, an erroneous decision to mistake a local minimum as a global minimum is prone to happen. Several noise levels are tested and it is observed that the PSO algorithm can find gBest accurately for noise levels up to 0.5. Once the noise level exceeds 0.5, the PSO algorithm begins to suffer from premature convergence to a local minima, especially at the noise level of 1. Therefore, the results are reported at two noise levels: 0.5 and 1.

Fig. 9 reports the simulation results for Rastrigin fitness function in three rows and two columns, similar to Fig. 6, except that the middle row in Fig. 9 is at a noise level of 0.5, and the bottom row is at a noise level of 1.

Without fitness evaluation noise, both PSO and NPSO achieve nearly perfect accuracy at the end of the iterations, but NPSO takes slightly more iterations to converge.

At the noise level of 0.5, the clustered distance between gBest and the true minimum clearly shows that NPSO often converges to the global minima (about 80%), but it may also converge outside of this minima the rest of the time.

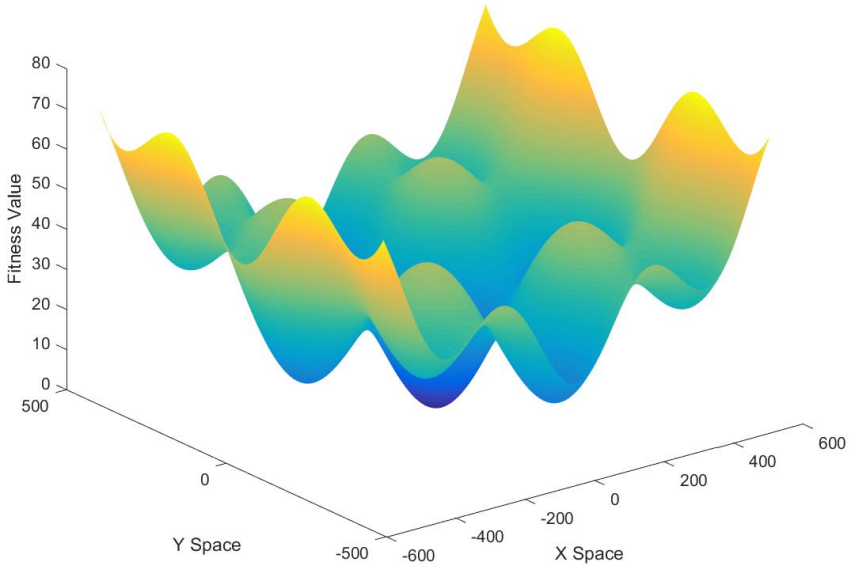


Figure 7: Surface plot of Scaled Rastrigin Function

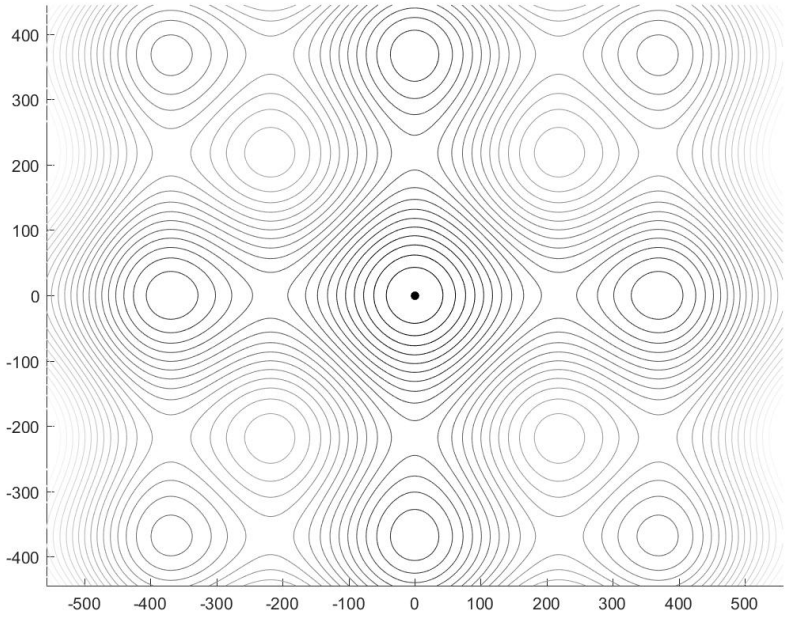


Figure 8: Contour plot of Scaled Rastrigin Function

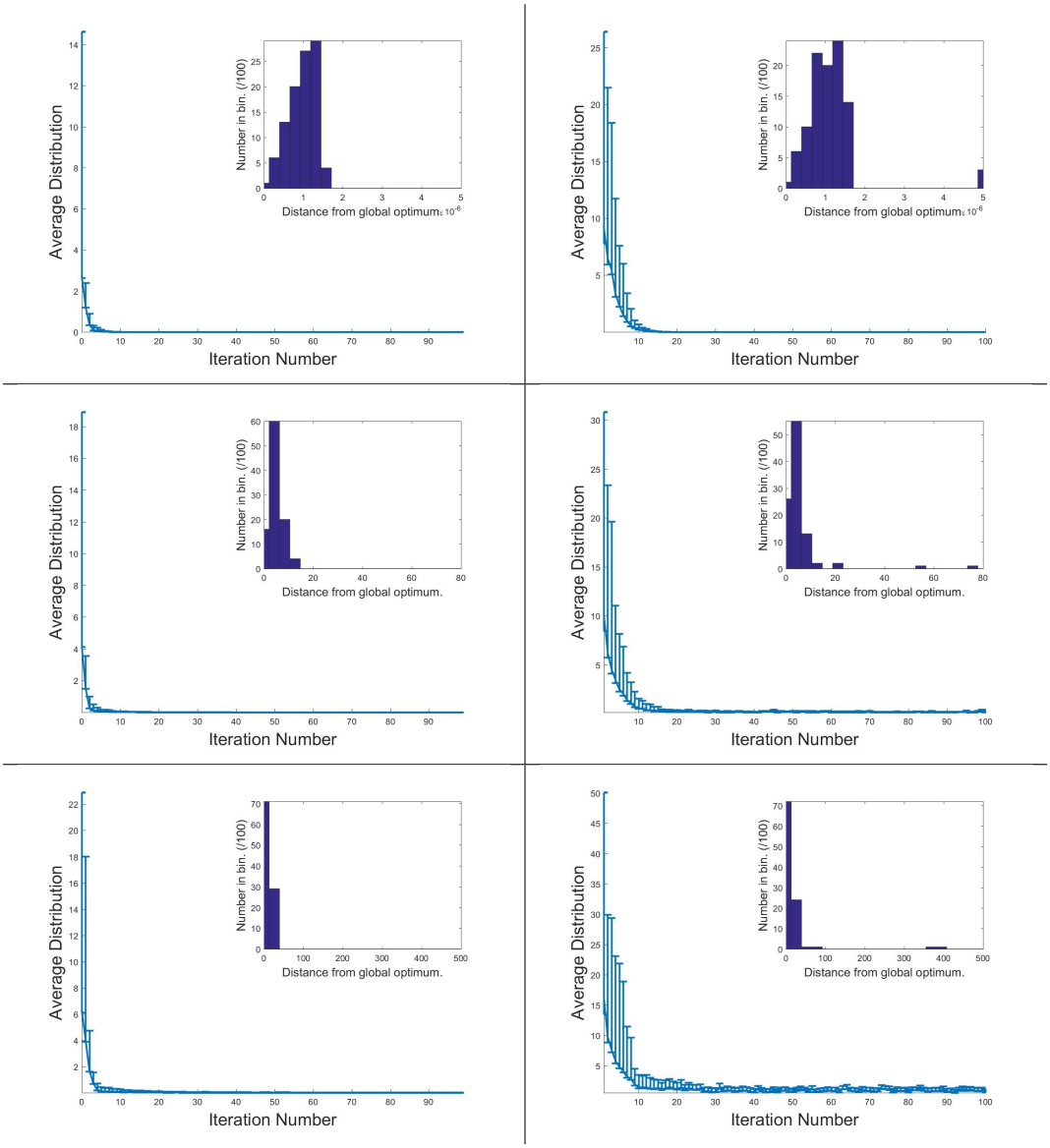


Figure 9: Results with a scaled Rastrigin fitness function. The performance of each scenario is reported in a fitness progression plot, and histogram of the distance between gBest and the true minimum. The left column uses PSO and the right column uses NPSO. The top plot of each column is without noise, the middle plot is at a noise level of 0.5, and the bottom plot is at a noise level of 1.

At the noise level of 1, PSO may also fall into a local minimum, but this happens more often by NPSO. The clustered distances in the plot in the bottom-row and the second column include 368, indicating that NPSO has mistaken a local minima as the optimum solution, although the majority is closer to the global minimum.

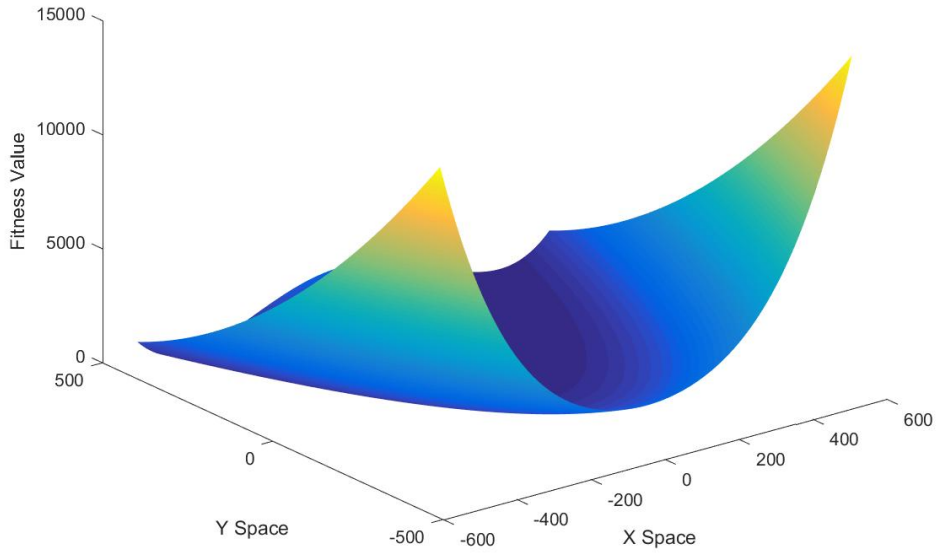


Figure 10: Surface plot of Scaled Rosenbrock function

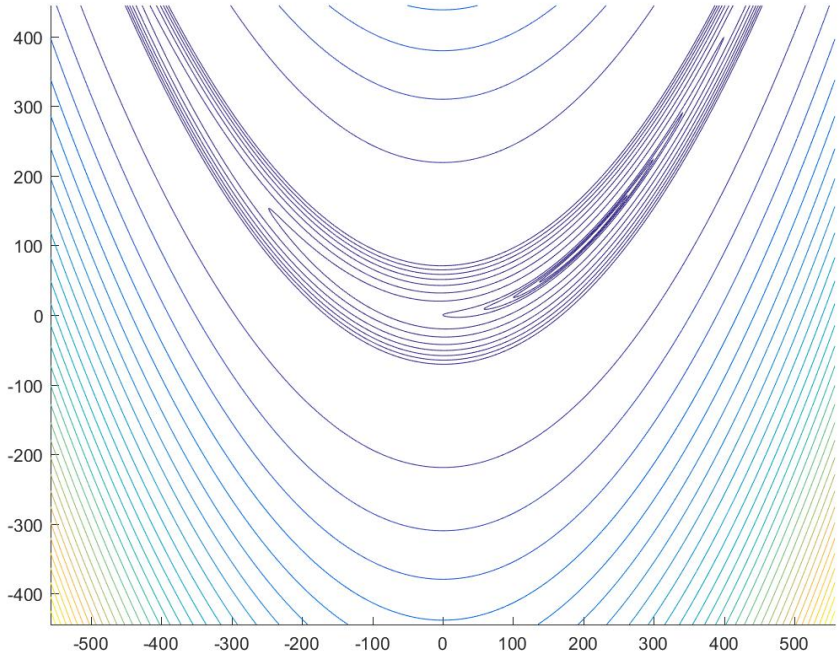


Figure 11: Contour plot of Scaled Rosenbrock function

4.1.3 Results Using Rosenbrock Fitness Functions

As analyzed in the previous two sub-sections, the spherical function is representative of a unimodal and symmetric fitness function, and the Rastrigin function is representative of a multi-modal and symmetric fitness function. In some other applications such as pipe-line fault detection, the fault is at certain point on the pipe, but the neighboring segments also often show degradation, and the degree of degradation typically is not symmetric. The Rosenbrock function, as defined in (4.3), is representative of such unimodal but asymmetric fitness function.

$$f(\mathbf{x}) = \sum_{j=1}^{d-1} [100(x_{j+1} - x_j^2)^2 + (x_j - 1)^2] \tag{4.3}$$

The surface and contour plots of a scaled 2D Rosenbrock function are shown in Fig. 10 and Fig. 11, respectively. The surface of the Rosenbrock function is in the shape of a shallow and asymmetric valley and minimizes at a single point. The long, thin contour groups show the valley, in the middle of which the global minimum lies.

Using the Rosenbrock fitness function, the same experimentation setup is used as that of the spherical function. The noise level is set at 0.25, since there is no fitness difference between the global minimum and local minima to compare to.

Fig. 12 reports the simulation results by a quad-chart, in the same way as in Fig. 6. The effect of noise is more obvious when the fitness surface near the global optimum is in the shape of a shallow valley than in the shape of a spherical bowl. This can be seen in Fig. 13, where the solutions fall within the valley. Although the displacements between the final solution and the global optimum appear large, the fitness values of those points differ by a little more than 0.25, indicating how challenging this problem is.

4.1.4 Performance Metrics

Quantitative performance metrics on the performance of PSO and NPSO under various noise levels are also evaluated:

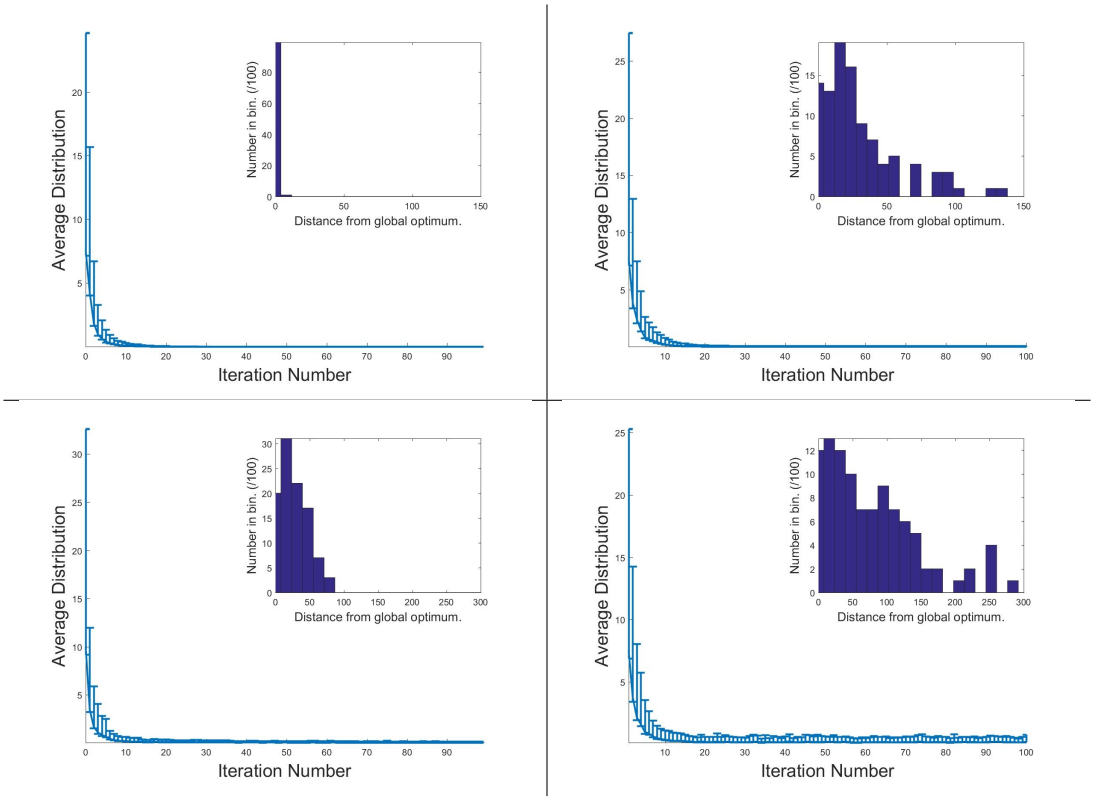
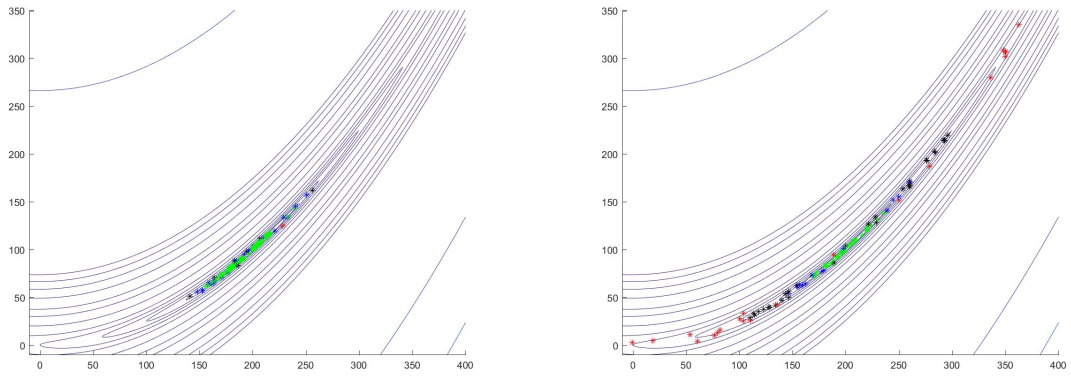


Figure 12: Results with a Rosenbrock fitness function. The performance of each scenario is reported in a fitness progression plot, and a histogram of the distance between gBest and the true minimum. The left column uses PSO, and the right column uses NPSO. The left plot in each column is without noise, and the right plot is at a noise level of 0.25.



(a) (b)

Figure 13: Final solutions for PSO and NPSO of Rosenbrock using noise of 0.25.

- Accuracy: the fitness difference between the best position and the global optimum at the end of the simulation. It is observable in the fitness progression plots. The lower the fitness progression line towards 0 (as the optimum solution's fitness is 0), the better the accuracy. The average, minimum and maximum values of the accuracy of all the simulated scenarios are presented in Table 4.1.

- Consistency: the square root of the squared difference between accuracy of each run and the average accuracy across all simulation runs. Intuitively, consistency shows the spread of the performance variation across different runs. As shown in the earlier plots, the smaller the distance between the 90 and 10 percentile bars on the fitness progression plot at each iteration step, or the more concentrated the spread of the histogram, the better the consistency. The consistency of all the simulated scenarios is presented in Table 4.2.

The accuracy and consistency metrics reported in Table 4.1 and 4.2 are consistent with the results in Sections 4.1.1, 4.1.2, and 4.1.3. The NPSO algorithm achieves the best performance in an application using a unimodal symmetric fitness function, even better than the PSO algorithm when noise is considered. In an application with local minima, NPSO may fall into the local minima and its performance variation is relatively high.

Note that the robots are not dimension-less particles, and hence, in practice, a success in search is when the robot touches the target, but not just aligned at the center of the target. The

Table 4.1: Accuracy metric of all the simulations. The closer to 0 (the fitness of the true optimum), the better.

Function	Noise		PSO	NPSO
Sphere	0	Max	3.2548×10^{-22}	5.8603×10^{-22}
		Avg	1.1946×10^{-23}	2.3173×10^{-23}
		Min	4.8682×10^{-27}	2.9361×10^{-26}
	0.25	Max	0.0516	0.0221
		Avg	0.0095	0.0034
		Min	1.102×10^{-4}	1.6093×10^{-6}
Rastrigin	0	Max	0	0.001
		Avg	0	1.0111×10^{-5}
		Min	0	0
	0.5	Max	0.2634	7.1787
		Avg	0.042	0.1431
		Min	1.0354×10^{-4}	1.5975×10^{-4}
	1	Max	1.127	16.5191
		Avg	0.1883	0.5885
		Min	5.6787×10^{-5}	4.7128×10^{-4}
Rosenbrock	0	Max	2.6134×10^{-4}	0.2451
		Avg	1.1532×10^{-5}	0.0230
		Min	2.4668×10^{-11}	3.1485×10^{-6}
	0.25	Max	0.2990	1.0712
		Avg	0.0395	0.1915
		Min	4.7881×10^{-4}	9.2880×10^{-5}

Table 4.2: Consistency metric of all the simulations. The smaller, the better.

Function	Noise	PSO	NPSO
Sphere	0	4.2921×10^{-22}	7.3316×10^{-22}
	0.25	0.0952	0.0404
Rastrigin	0	0	0.001
	0.5	0.4598	7.9551
	1	1.9662	23.2066
Rosenbrock	0	3.8098×10^{-4}	0.4240
	0.25	0.4674	2.4220

histogram of the distance between the final gBest/nBest and the true optimum is reported, and when an application-dependent threshold of a successful detection is applied on the histograms,

success rate is readily available.

The convergence speed is critical for a timely completion of a mission. It is observed that the convergence speed is higher in unimodal functions (symmetric or asymmetric) than in the multi-modal function. In this thesis, the goal is to locate the global optimum, so the detection of local optima is counted unfavorably in the accuracy evaluation. In some other applications, however, if the identification of local optima is also of value, the exploration of the particles is advantageous.

4.2 Simulation Results Using Four Stages of Pseudo-Vector Motion

The results presented in this section is obtained from Kilombo simulations.

4.2.1 Results Using Spherical Well

Testing the algorithm begins by providing it a very basic fitness function. The fitness function should represent a task that would be interesting in a real-world setting. In this case, we simulate the swarm attempting to locate a fire event. The fire will be represented by the light source. In a real event, more than just ambient light sensors can be used to quantify the fitness of each Kilobot, giving more dimensions to the fitness. It would be optimal for the swarm to locate the event as quickly as possible. The fitness function here represents received light intensity based on the Kilobots position. In the instance of a single event, we can describe the function as a parabolic well with a single minimum, defined in (1). This function is provided as the default method to get ambient light intensity in the Kilombo simulator.

$$f(\mathbf{x}) = \sum_{j=1}^{d} \frac{x_j^2}{1000} \tag{4.4}$$

where $\mathbf{x}$ in $f(\mathbf{x})$ is the particle position. d is the dimension of the fitness function, in this case, 2, since we are searching a 2D space, but could be larger for more complex fitness, such adding ambient light and temperature. x_j is the jth coordinate of x, or x's position in the jth dimension. This notation is used similarly in (4.1, 4.2). In this thesis, fitness functions are bound to the realistic search space of our whiteboard (the recommended surface for the Kilobots to move on) to

evaluate the physical deployment, thus measurements are in millimeters. The parameters for this experiment are as follows: the swarm of 30 Kilobots will be bound in a 2D search space that is 1113mm by 889mm, the size of a 4 ft by 3 ft whiteboard. Each Spherical simulation will run for 500 seconds, which equates to 15501 kiloticks for the Kilobots, or 12000 steps in the simulator. We will take 101 unique samples during the course of the simulation, one at the beginning and one every 120 simulator steps.

Figure 14 reports the results from the simulations by quad- chart. In each quadrant the convergence of each scenario is reported by two fitness progression plots. These plots represent the variations amongst the 100 tests of each algorithm.

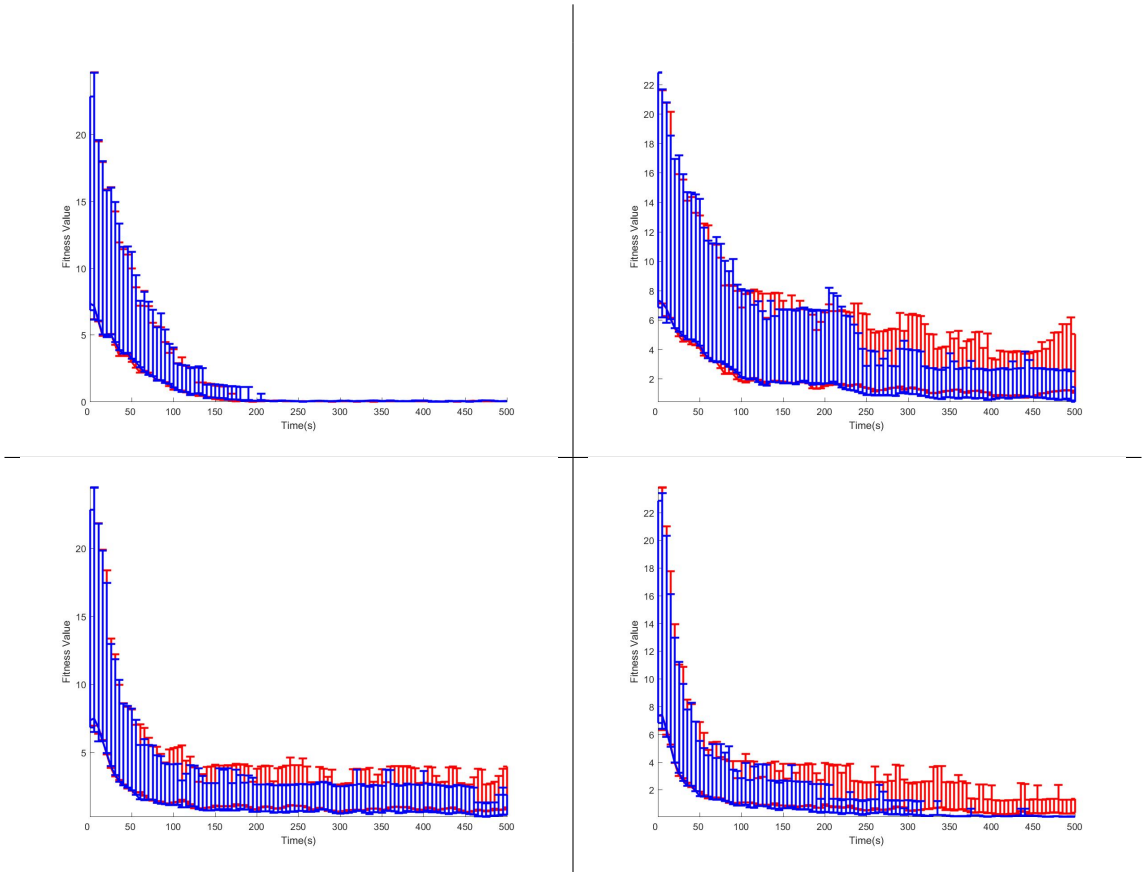


Figure 14: Results with the parabolic well fitness function. Blue is tested under ideal conditions, i.e. both messaging noise and distance noise are 0. Red is tested under the worst conditions, i.e. messaging noise is 0.8 and distance noise is 10mm. Top left reports the PVM, top right reports CPVM, bottom left reports S-CPVM and bottom right reports CS-CPVM.

The results in blue are of each algorithm under perfect conditions, no distance noise between the Kilobots and no dropped messages. The algorithms were then tested with both distance and messaging noise. The results under the worst conditions, with a distance measurement noise of 10 mm and a 20% dropping rate of the messages, are reported in red. The quicker it gets down to 0, the quicker the Kilobots converge on the minimum. However, just because a bot measured the minimum does not mean it registered that minimum as the true minimum. This can be observed in the simulator, especially in the first algorithm, when the bots swarm around the minimum but few actually stop inside it. For this reason, we also count the number of success found amongst the Kilobots. With 30 Kilobots to sample over the 100 Monte-Carlo simulations, that gives us 3000 samples for each algorithm. These success numbers are reported in Table 4.3. Using both pieces of information, we can see that although PVM appears to converge fastest, its stopping condition is not robust enough to register the event, following the observed behavior. We can also see that although there are almost as many successes in CPVM as there are in CS-CPVM, from the top-right graph in Figure 14 we can see that many of these registered successes are likely not the true minimum.

4.2.2 Results Using Rastrigin Fitness Functions

The parabolic well models a single event scenario well, but multi-event scenarios are possible, too. In this case, it may be useful to locate every event, but it is still most important to locate the primary event. This type of multi-modal searching becomes more challenging. In order to test this case, a second fitness function is provided to compare the functionality in a multi-event environment. For this instance, we use a modified Rastrigin function as defined in (4.5).

$$f(\mathbf{x}) = 10d + \sum_{j=1}^{d} [\frac{x_j^2}{100} - 10\cos(\frac{\pi}{2}\frac{x_j}{100})] \tag{4.5}$$

Note that the scale of this function was adapted to keep the measurements within the observable range of the hardware. In addition, we reduce the phase of the Rastrigin function from 2π to $\pi/2$, in order to reduce the number of minima. Under these conditions there will exist three distinct values across nine minima laid out in a checkerboard pattern.

The distribution of the 3000 samples (30 Kilobots in 100 Monte Carlo runs) across these nine minima is reported in Figure 15. Each bar in Figure 15 represents the number of bots in that minima, with the blue part registered successes within two bots distance of the minima, the green part registered successes outside two bots distance from the minima and the yellow part bots that have not yet registered a success.

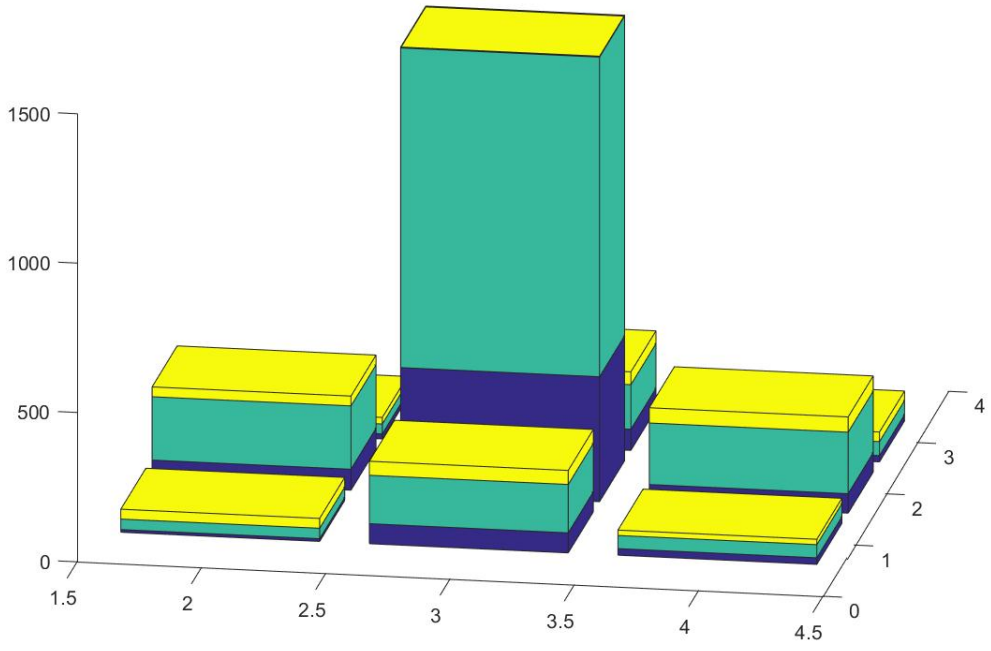


Figure 15: Distribution of the locations of all the found solutions by all 3000 Kilobots (30 bots in 100 Monte-Carlo runs)

The middle column in Figure 15 is the global minimum, or the 1st level minimum, corresponding to the center of Figures 7 and 8. The majority of the Kilobots are landing there. The four 2nd level local minima are at the horizontal and vertical lines passing the center in Figures 7, 8, and 15. The four 3rd level local minima are at the farthest diagonal locations in Figures 7, 8,

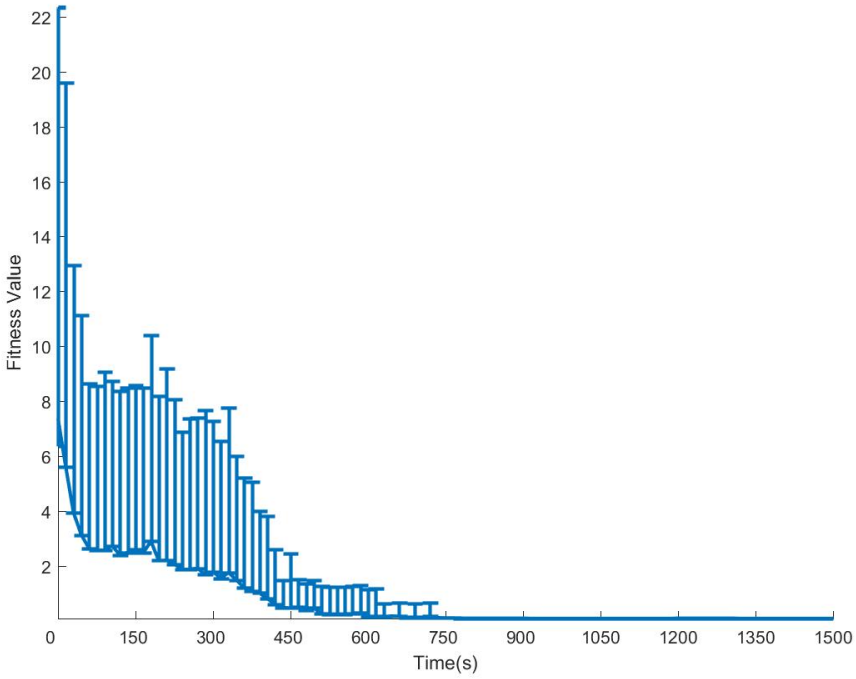


Figure 16: Convergence with Rastrigin Function.

and 15. The population of Kilobots landing in the 2nd level minima are much fewer than those landing in the 1st level minimum, but are much more than those landing in the 3rd level minima. The bots that have not yet registered a success (in yellow in Figure 15) are few in number.

The convergence of this simulation under perfect conditions is reported in Figure 16. It converges slower, with simulations running for 1500 seconds, three times as long as the uni-modal case. This is most likely due to the steeper difference in fitness in this function and the saddles created between minima and maxima. However, it also registered a similar percentage of success as the uni-modal case, which may be an indication of robustness in the face of multiple events.

Table 4.3: Success out of 3000 samples.

Algorithm	Noise Source	# of Success	Success Percentage
PVM	None	160	5.330%
	Distance	136	4.53%
	Measurement	151	5.03%
	Both	164	5.47%
CPVM	None	2630	87.666%
	Distance	2247	74.9%
	Measurement	2309	76.97%
	Both	2270	75.67%
S-CPVM	None	1205	40.166%
	Distance	941	31.37%
	Measurement	1188	39.6%
	Both	1007	33.57%
CS-CPVM	None	2685	89.5%
	Distance	2283	76.1%
	Measurement	2542	84.73%
	Both	2219	73.97%

4.2.3 Success Metric

The results are presented in convergence plots between the best and the worst conditions, and the success rates are reported in both total counting out of 3000 Kilobot samples and the claimed successes' percentage. Note that the success is defined to be either at the location of the best solution (the minimum of the fitness function), or the Kilobot is next to a Kilobot that has claimed to have registered success. Although the success rate of CS-CPVM appears a bit lower than that of CPVM, the fitness values of the CS-CPVM convergence results are much closer to the true solution and with much smaller variance, and hence CS-CPVM is the best algorithm of all.

CHAPTER 5: Conclusion and Future Work

This thesis studied Particle Swarm Optimization (PSO) as a swarm intelligence technique and its effectiveness in control of a Kilobot swarm. It started by adapting PSO to the physical constraints of the Kilobots through Matlab simulations, then adapted the resultant Neighborhood PSO (N-PSO) algorithm into a pseudo-vector motion robotic control based on the capabilities of the bots in the Linux-based Kilombo simulation environment.

NPSO was tested against the canonical PSO, and both were tested under the presence of measurement noise. The NPSO algorithm was shown to be robust enough to locate solutions in a given search space almost as well as PSO. In fact, some cases of our adapted algorithm converged upon a solution quicker when under the effects of noise versus canonical PSO, as seen in the Monte-Carlo simulation results for NPSO. Further physical limitations required more adaptation into the robotic control, namely, a continuous method to replace the vector motion is needed.

The adapted Pseudo-Vector Motion (PVM) algorithm, incorporated into the Kilobots, is not exactly the same as PSO, but the idea to incorporate momentum, self-cognizance, and social influence, in guiding the Kilobot movements in PVM is the same as in canonical PSO and NPSO. PVM addressed the physical constraints of Kilobots making it one big step forward to control the real Kilobots. The evaluation metrics of all the proposed schemes include how quickly and how efficiently the search tasks are done on two benchmark functions.

In simulations and in foreseeable physical experiments, once some robots have found the best location, other robots may still try to get into that area and tend to push the earlier robots out. As a result, we proposed a success-declaration scheme where once some robots have declared success, other neighboring robots will subsequently declare success and stay nearby instead of pushing out earlier robots.

The PVM-based algorithms were developed and enhanced in four stages (PVM, CPVM, S-CPVM, and CS-CPVM). The four stages differ in the number of factors to guide the movement of Kilobots, as well as the success declaration scheme when the robots are about to finish their searching. The movement guiding factors added at each new stage are in effect with the previous factors.

The fist stage is PVM, representing the usage of momentum alone. The second stage is Cognitive PVM (CPVM), where Kilobots remember how they have been moving, adding a cognitive factor. The third stage is Social-CPVM (S-CPVM), where Kilobots not only remember where they themselves have been, but also communicate with neighbors to know where their neighbors' best locations have been, adding a social influence factor. In the fourth stage, the Cumulative Social-Cognitive Pseudo-Vector Motion (CS-CPVM), the movement guidance is similar to the third stage (S-CPVM), but the success declaration scheme is updated to count the cumulative duration for a bot to stay in one region to make the declaration more reliable.

In this thesis, three benchmark functions are used representing various degrees of search difficulty, with one or more local optima, and with distinct or negligible differences around the optima. In the future, other benchmark functions representing even more challenging search environment can be explored.

Meanwhile, the four stages of PVM algorithms are determined in nature using rules instead of using a random weight to combine the effects due to momentum, self-cognizance and social influence factors. In the future, we plan to incorporate a random scheme, such as a roulette wheel random selection procedure, to combine the three factors to guide the movement of Kilobots.

The code developed in Kilombo for this thesis has included some new data structure definitions, and we are exploring the portability of the code onto real Kilobots processors. Implementing the control algorithms with physical measurements (such as light intensity, temperature, and volume) instead of artificial call-back functions will be very exciting.

In Kilombo simulations, the Kilobots' speed and turning rate are held at certain value, and the values of the physical Kilobots could deviate from those, yielding both a challenge in

code consistency and an opportunity to have a wider range of parameters to tune to improve the robustness of the current system.

Bibliography

[1] Hyondong Oh, Ataollah R. Shiraz, and Yaochu Jin, "Morphogen diffusion algorithms for tracking and herding using a swarm of kilobots," *Soft Computing*, pp. 1–12, 2016.

[2] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *1995 IEEE International Conference on Neural Networks*, Nov 1995, vol. 4, pp. 1942–1948.

[3] M. Rubenstein, C. Ahler, and R. Nagpal, "Kilobot: A low cost scalable robot system for collective behaviors," in *2012 IEEE International Conference on Robotics and Automation (ICRA)*, May 2012, pp. 3293–3298.

[4] R. Eberhart and Y. Shi, "Particle swarm optimization: developments, applications and resources," in *Proceeding of the 2001 Congress on Evolutionary Computation*, 2001, vol. 1, pp. 81–86 vol. 1.

[5] Millie Pant, Radha Thangaraj, and Ajith Abraham, *Foundations of Computational Intelligence Volume 3: Global Optimization*, chapter Particle Swarm Optimization: Performance Tuning and Empirical Analysis, pp. 101–128, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[6] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, May 1998, pp. 69–73.

[7] Neveen I Ghali, Nahed El-Dessouki, AN Mervat, and Lamiaa Bakrawi, "Exponential particle swarm optimization approach for improving data clustering," *International Journal of Electrical, Computer, and Systems Engineering*, vol. 3, no. 4, pp. 208–212, 2009.

[8] Alireza ALFI, "{PSO} with adaptive mutation and inertia weight and its application in parameter estimation of dynamic systems," *Acta Automatica Sinica*, vol. 37, no. 5, pp. 541 – 549, 2011.

[9] J. M. Hereford, M. Siebold, and S. Nichols, "Using the particle swarm optimization algorithm for robotic search applications," in *2007 IEEE Swarm Intelligence Symposium (SIS)*, April 2007, pp. 53–59.

[10] E. Di Mario, I. Navarro, and A. Martinoli, "Analysis of fitness noise in particle swarm optimization: From robotic learning to benchmark functions," in *2014 IEEE Congress on Evolutionary Compuation (CEC)*, July 2014, pp. 2785–2792.

[11] R. Eberhart and Y. Shi, "Comparing inertia weights and constriction factors in particle swarm optimization," in *Proceedings of the 2000 Congress on Evolutionary Computation*, 2000, vol. 1, pp. 84–88 vol.1.

[12] J. W. Lee and J. J. Lee, "Gaussian-distributed particle swarm optimization: A novel gaussian particle swarm optimization," in *2013 IEEE International Conference on Industrial Technology (ICIT)*, Feb 2013, pp. 1122–1127.

[13] Christopher W. Cleghorn and Andries P. Engelbrecht, "Particle swarm variants: standardized convergence analysis," *Swarm Intelligence*, vol. 9, no. 2, pp. 177–203, 2015.

[14] Ahmad Nickabadi, Mohammad Mehdi Ebadzadeh, and Reza Safabakhsh, "A novel particle swarm optimization algorithm with adaptive inertia weight," *Applied Soft Computing*, vol. 11, no. 4, pp. 3658 – 3670, 2011.

[15] Bin Jiao, Zhigang Lian, and Xingsheng Gu, "A dynamic inertia weight particle swarm optimization algorithm," *Chaos, Solitons & Fractals*, vol. 37, no. 3, pp. 698 – 705, 2008.

[16] M. Pant, R. Thangaraj, C. Grosan, and A. Abraham, "Improved particle swarm optimization with low-discrepancy sequences," in *Evolutionary Computation, 2008. CEC 2008. (IEEE*

World Congress on Computational Intelligence). IEEE Congress on, June 2008, pp. 3011–3018.

[17] M. Pant, T. Radha, and V.P. Singh, "A new diversity based particle swarm optimization using gaussian mutation," *Int. J. of Mathematical Modeling, Simulation and Applications*, vol. 1(1), pp. 47–60, 2008.

[18] K. E. Parsopoulos and M. N. Vrahatis, "Particle swarm optimizer in noisy and continuously changing environments," in *M.H. Hamza (Ed.), Artificial Intelligence and Soft Computing, IASTED/ACTA*. 2001, pp. 289–294, IASTED/ACTA Press.

[19] Thomas Bartz-Beielstein, Daniel Blum, and Jürgen Branke, *Metaheuristics: Progress in Complex Systems Optimization*, chapter Particle Swarm Optimization and Sequential Sampling in Noisy Environments, pp. 261–273, Springer US, Boston, MA, 2007.

[20] Hui Pan, Ling Wang, and Bo Liu, "Particle swarm optimization for function optimization in noisy environment," *Applied Mathematics and Computation*, vol. 181, no. 2, pp. 908 – 919, 2006.

[21] Y. Nakano and H. Takagi, "Influence of fitness quantization noise on the performance of interactive pso," in *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, May 2009, pp. 2416–2422.

[22] M. Freese E. Rohmer, S. P. N. Singh, "V-rep: a versatile and scalable robot simulation framework," in *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[23] Antti Halme, "Kilobot app - a kilobot simulator and swarm pattern designer," 2012.

[24] Fredrik Jansson, Matthew Hartley, Martin Hinsch, Ivica Slavkov, Noemí Carranza, Tjelvar S. G. Olsson, Roland M. Dries, Johanna H. Grönqvist, Athanasius F. M. Marée, James Sharpe, Jaap A. Kaandorp, and Verônica A. Grieneisen, "Kilombo: a kilobot simulator to enable effective research in swarm robotics," *CoRR*, vol. abs/1511.04285, 2015.

[25] J. Kennedy, "The particle swarm: social adaptation of knowledge," in *Evolutionary Computation, 1997., IEEE International Conference on*, Apr 1997, pp. 303–308.

[26] P. N. Suganthan, "Particle swarm optimiser with neighbourhood operator," in *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, 1999, vol. 3, p. 1962 Vol. 3.

[27] Xiaohui Hu and R. Eberhart, "Multiobjective optimization using dynamic neighborhood particle swarm optimization," in *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, 2002, vol. 2, pp. 1677–1681.

[28] T. Ali, S. Anis, and M. Faouzi, "Multi-objective predictive control using discrete ts fuzzy systems and a modified dynamic neighborhood pso," in *Systems, Signals Devices (SSD), 2015 12th International Multi-Conference on*, March 2015, pp. 1–6.

[29] M. Stender, Y. Yan, H. B. Karayaka, P. Tay, and R. Adams, "Simulating micro-robots to find a point of interest under noise and with limited communication using particle swarm optimization," in *SoutheastCon 2017*, March 2017, pp. 1–8.

[30] J. Vesterstrom and R. Thomsen, "A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems," in *Evolutionary Computation, 2004. CEC2004. Congress on*, June 2004, vol. 2, pp. 1980–1987 Vol.2.

APPENDIX A: Particle Swarm Optimization Test Results

A.1 Sphere Figures for PSO

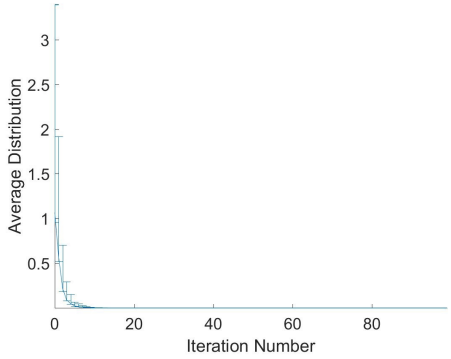


Figure 17: Convergence PSO using Spherical Fitness and fitness noise = 0

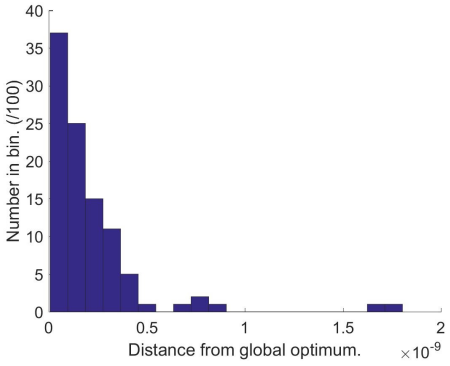


Figure 18: Distance from true minimum for PSO using Spherical fitness and fitness noise = 0

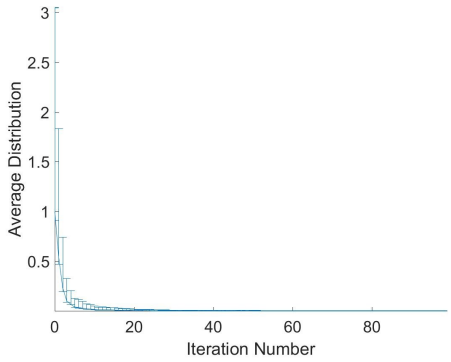


Figure 19: Convergence of PSO using Spherical Fitness and fitness noise = 0.1

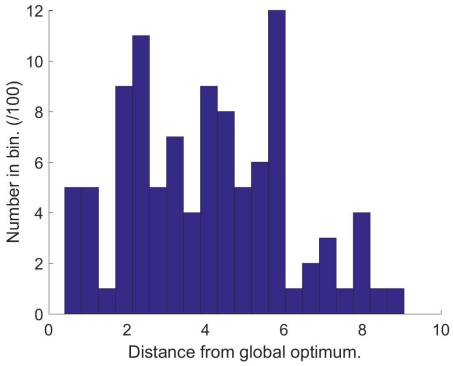


Figure 20: Distance from true minimum for PSO using Spherical fitness and fitness noise = 0.1

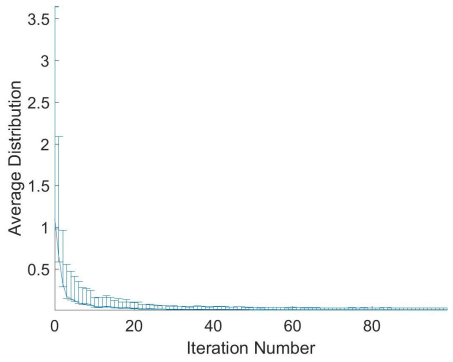


Figure 21: Convergence of PSO using Spherical Fitness and fitness noise = 0.25

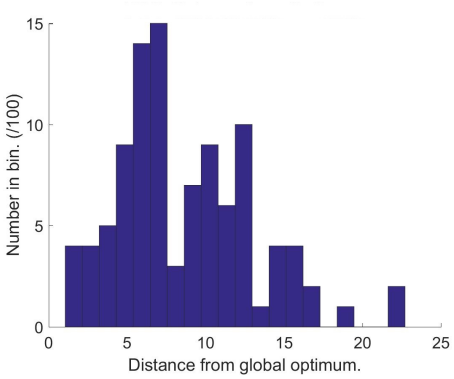


Figure 22: Distance from true minimum for PSO using Spherical fitness and fitness noise = 0.25

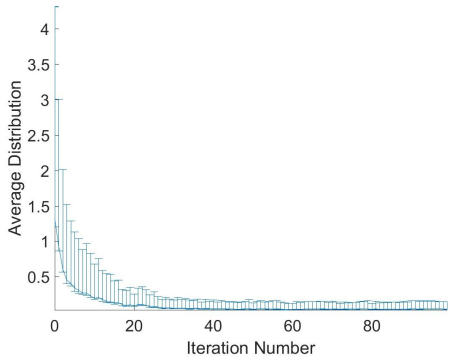


Figure 23: Convergence of PSO using Spherical Fitness and fitness noise = 0.5

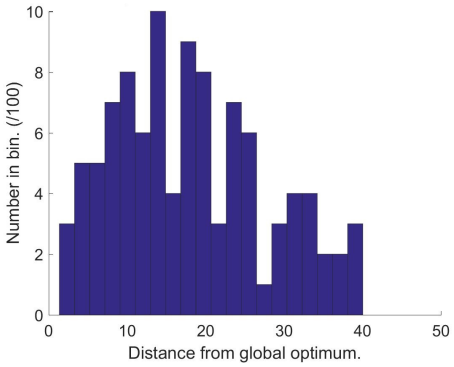


Figure 24: Distance from true minimum for PSO using Spherical fitness and fitness noise = 0.5

A.2 Rastrigin Figures for PSO

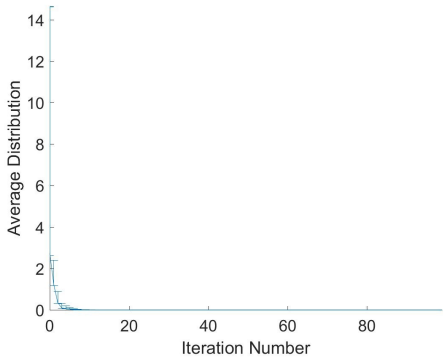


Figure 25: Convergence of PSO using Rastrigin Fitness and fitness noise = 0

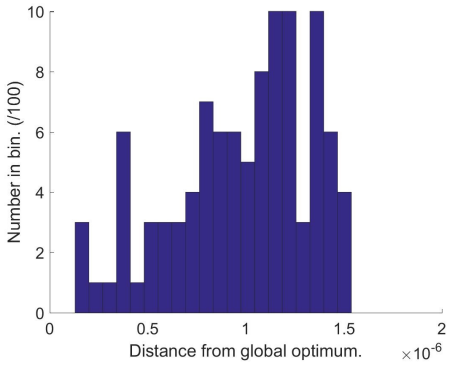


Figure 26: Distance from true minimum for PSO using Rastrigin fitness and fitness noise = 0

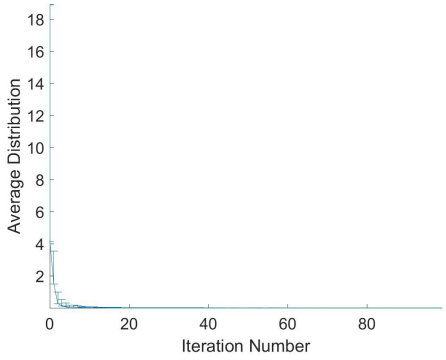


Figure 27: Convergence of PSO using Rastrigin Fitness and fitness noise = 0.5

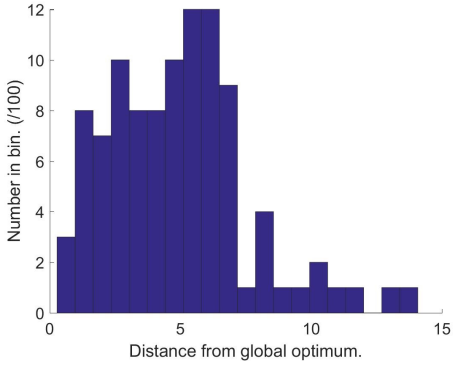


Figure 28: Distance from true minimum for PSO using Rastrigin fitness and fitness noise = 0.5

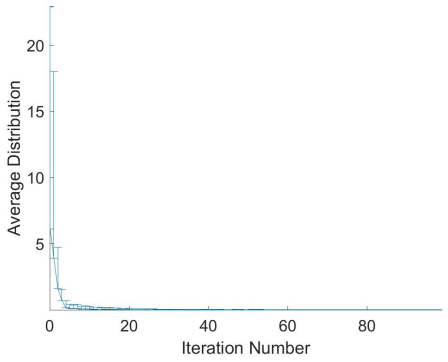


Figure 29: Convergence of PSO using Rastrigin Fitness and fitness noise = 1.0

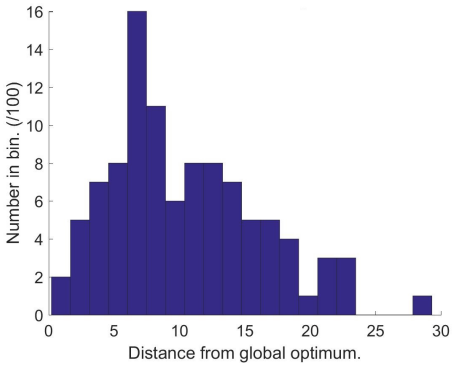


Figure 30: Distance from true minimum for PSO using Rastrigin fitness and fitness noise = 1.0

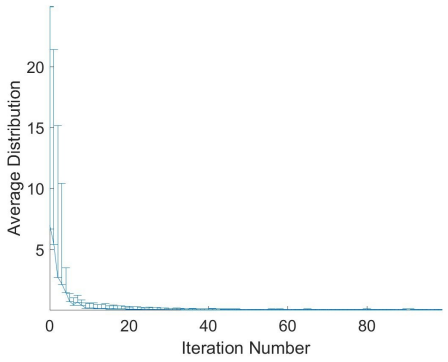


Figure 31: Convergence of PSO using Rastrigin Fitness and fitness noise = 1.5

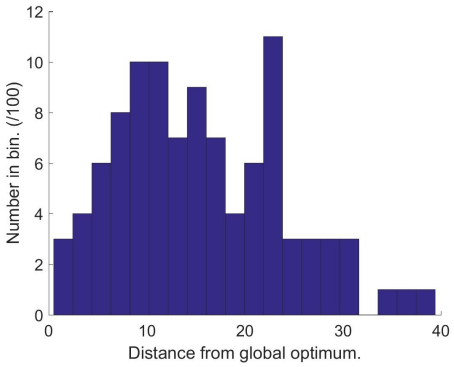


Figure 32: Distance from true minimum for PSO using Rastrigin fitness and fitness noise = 1.5

A.3 Rosenbrock Figures for PSO

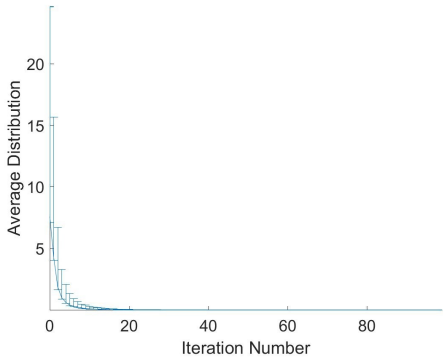


Figure 33: Convergence of PSO using Rosenbrock Fitness and fitness noise = 0

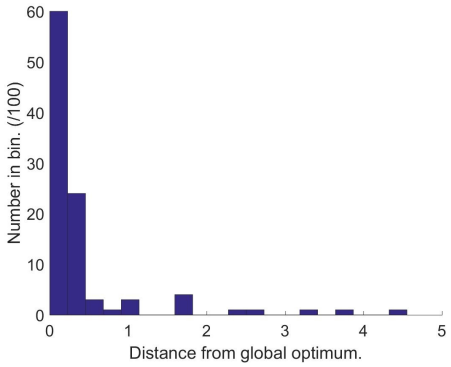


Figure 34: Distance from true minimum for PSO using Rosenbrock fitness and fitness noise = 0

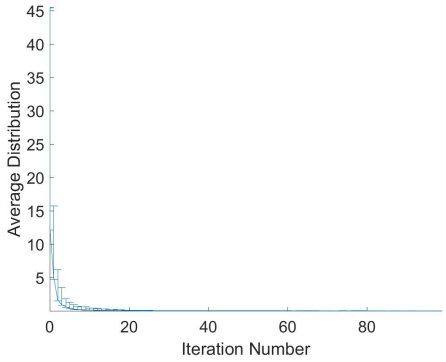


Figure 35: Convergence of PSO using Rosenbrock Fitness and fitness noise = 0.1

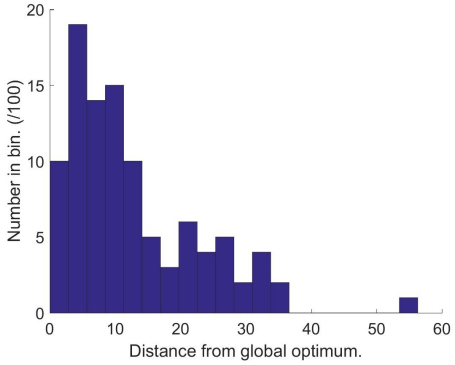


Figure 36: Distance from true minimum for PSO using Rosenbrock fitness and fitness noise = 0.1

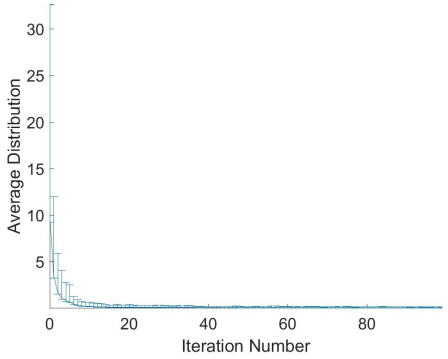


Figure 37: Convergence of PSO using Rosenbrock Fitness and fitness noise = 0.25

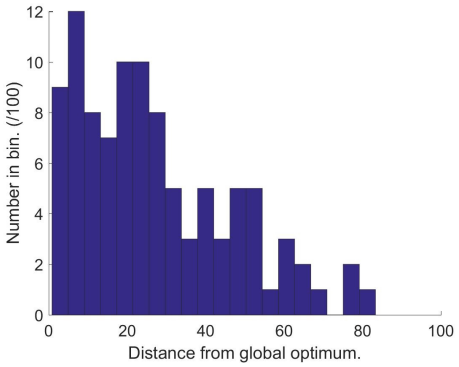


Figure 38: Distance from true minimum for PSO using Rosenbrock fitness and fitness noise = 0.25

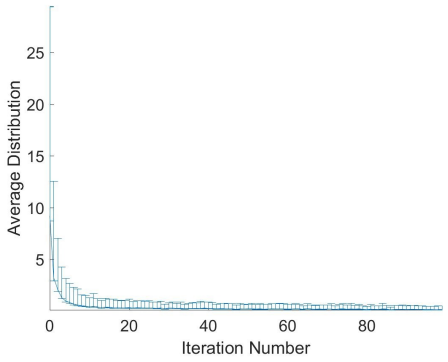


Figure 39: Convergence of PSO using Rosenbrock Fitness and fitness noise = 0.5

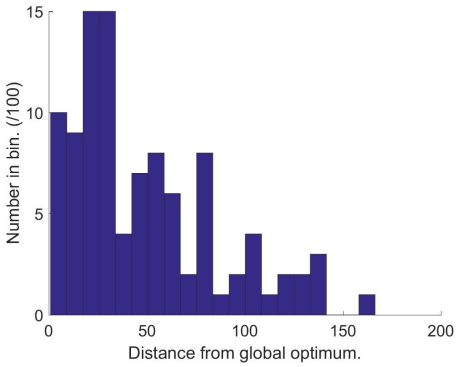


Figure 40: Distance from true minimum for PSO using Rosenbrock fitness and fitness noise = 0.5

A.4 Sphere Figures for NPSO

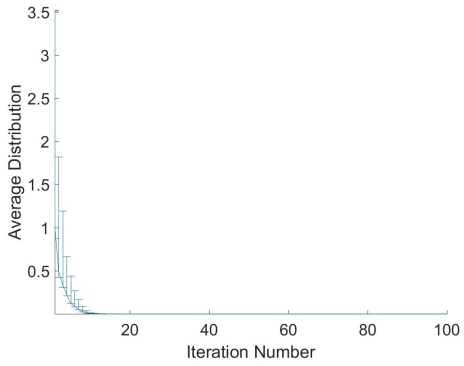


Figure 41: Convergence of NPSO using Spherical Fitness and fitness noise = 0

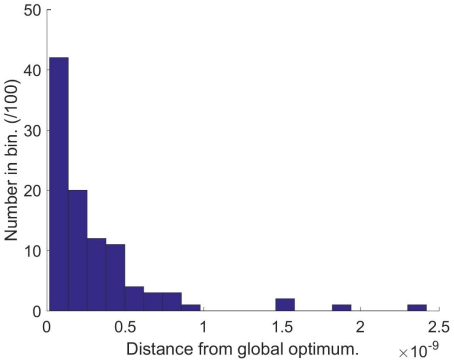


Figure 42: Distance from true minimum for NPSO using Spherical fitness and fitness noise = 0

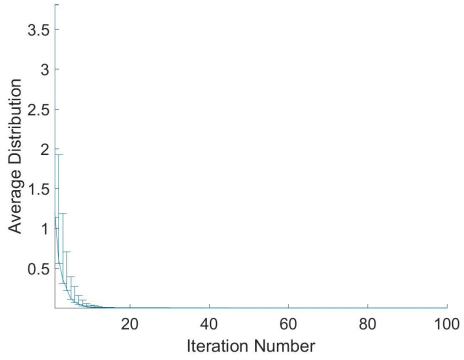


Figure 43: Convergence of NPSO using Spherical Fitness and fitness noise = 0.1

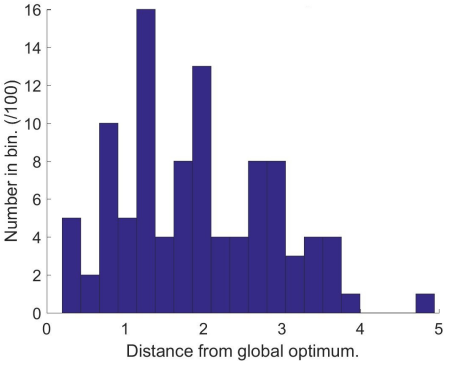


Figure 44: Distance from true minimum for NPSO using Spherical fitness and fitness noise = 0.1

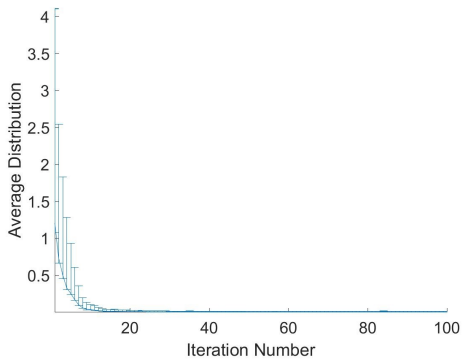


Figure 45: Convergence of NPSO using Spherical Fitness and fitness noise = 0.25

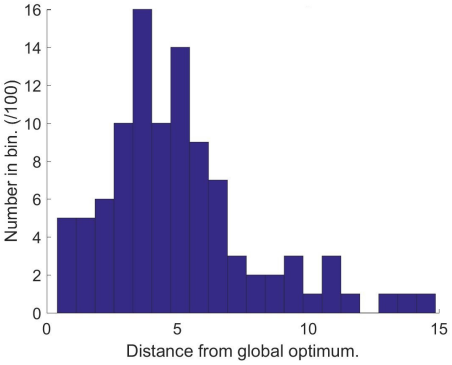


Figure 46: Distance from true minimum for NPSO using Spherical fitness and fitness noise = 0.25

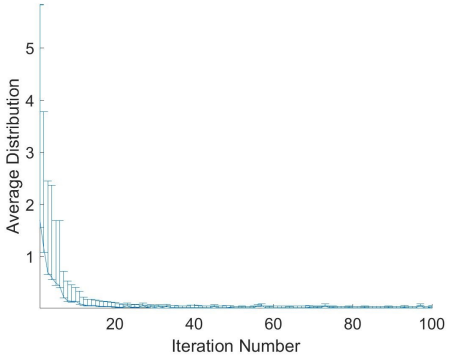


Figure 47: Convergence of NPSO using Spherical Fitness and fitness noise = 0.5

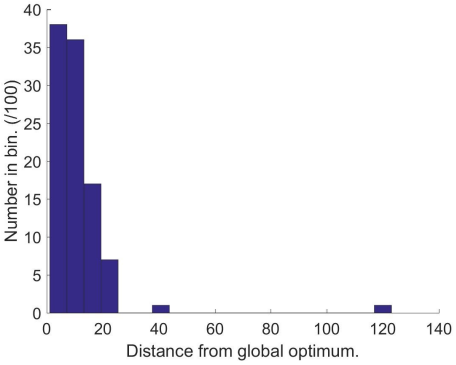


Figure 48: Distance from true minimum for NPSO using Spherical fitness and fitness noise = 0.5

A.5 Rastrigin Figures for NPSO

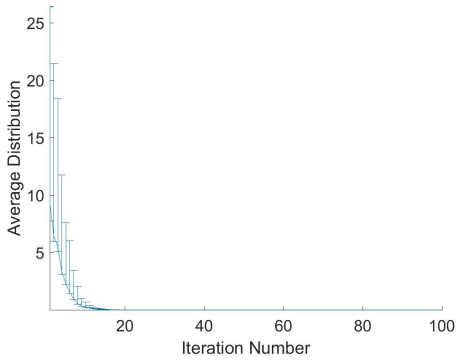


Figure 49: Convergence of NPSO using Rastrigin Fitness and fitness noise = 0

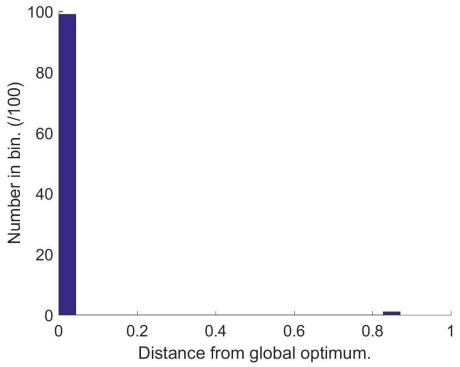


Figure 50: Distance from true minimum for NPSO using Rastrigin fitness and fitness noise = 0

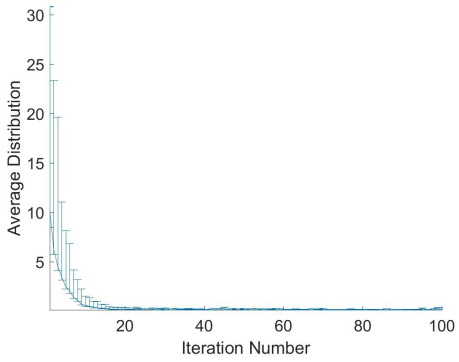


Figure 51: Convergence of NPSO using Rastrigin Fitness and fitness noise = 0.5

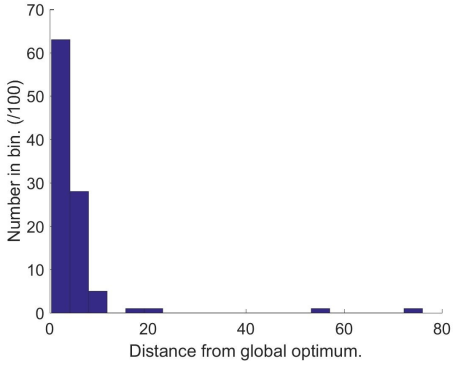


Figure 52: Distance from true minimum for NPSO using Rastrigin fitness and fitness noise = 0.5

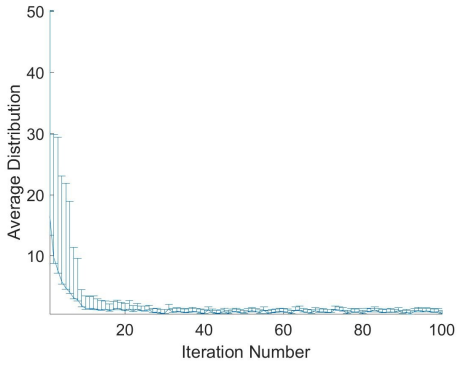


Figure 53: Convergence of NPSO using Rastrigin Fitness and fitness noise = 1.0

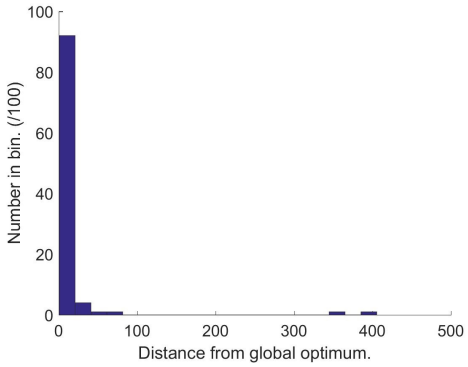


Figure 54: Distance from true minimum for NPSO using Rastrigin fitness and fitness noise = 1.0

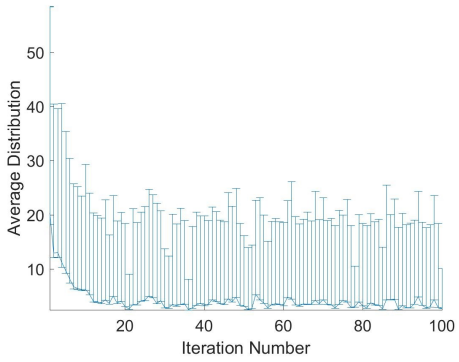


Figure 55: Convergence of NPSO using Rastrigin Fitness and fitness noise = 1.5

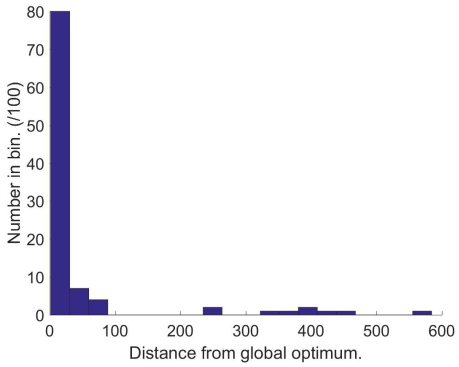


Figure 56: Distance from true minimum for NPSO using Rastrigin fitness and fitness noise = 1.5

A.6 Rosenbrock Figures for NPSO

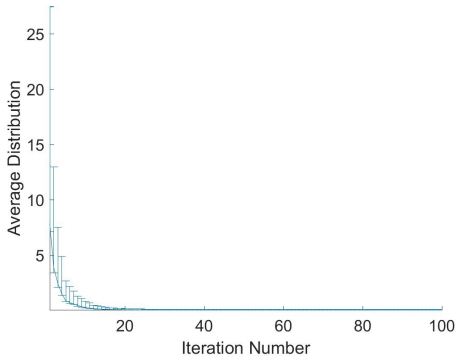


Figure 57: Convergence of NPSO using Rosenbrock Fitness and fitness noise = 0

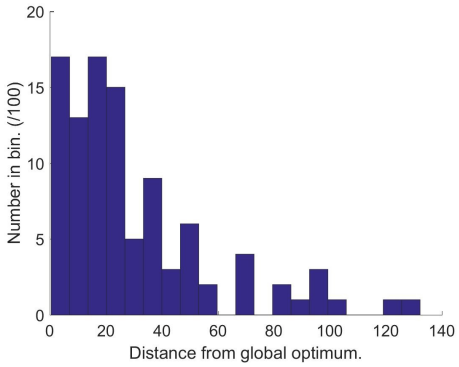


Figure 58: Distance from true minimum for NPSO using Rosenbrock fitness and fitness noise = 0

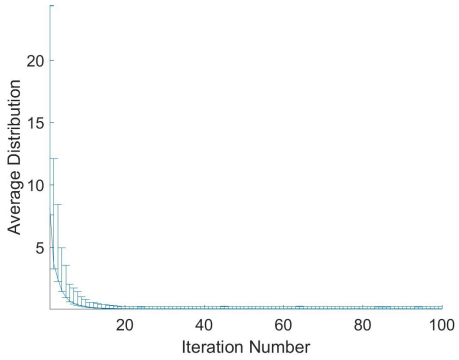


Figure 59: Convergence of NPSO using Rosenbrock Fitness and fitness noise = 0.1

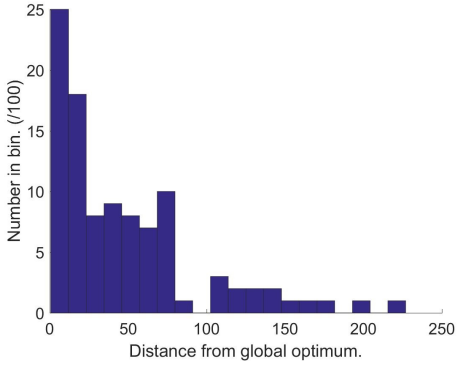


Figure 60: Distance from true minimum for NPSO using Rosenbrock fitness and fitness noise = 0.1

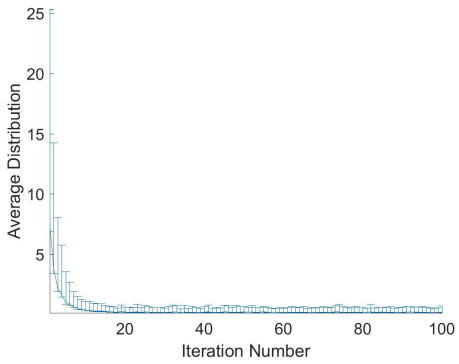


Figure 61: Convergence of NPSO using Rosenbrock Fitness and fitness noise = 0.25

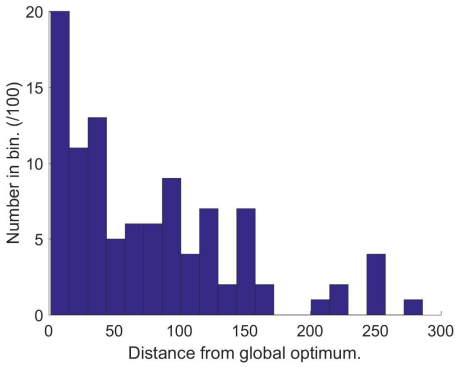


Figure 62: Distance from true minimum for NPSO using Rosenbrock fitness and fitness noise = 0.25

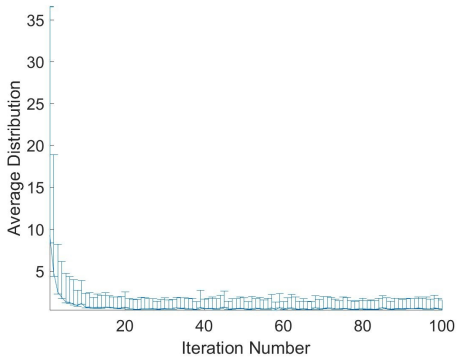


Figure 63: Convergence of NPSO using Rosenbrock Fitness and fitness noise = 0.5

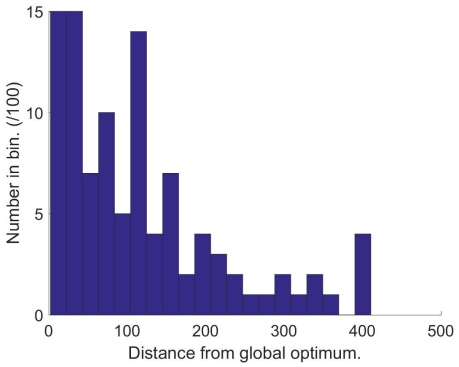


Figure 64: Distance from true minimum for NPSO using Rosenbrock fitness and fitness noise = 0.5

APPENDIX B: Kilombo Simulator Test Results

B.1 Pseudo Vector Motion (PVM)

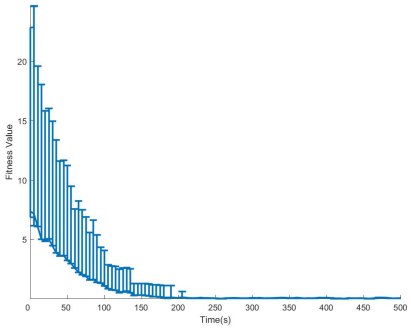


Figure 65: PVM using Spherical Fitness, Distance Noise = 0mm, Messaging Success 100%

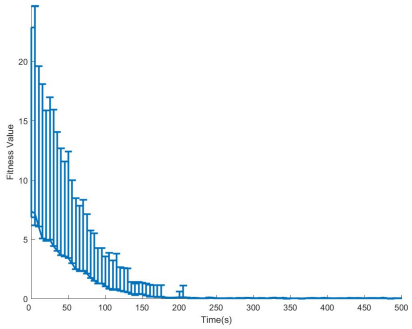


Figure 66: PVM using Spherical Fitness, Distance Noise = 0mm, Messaging Success 80%

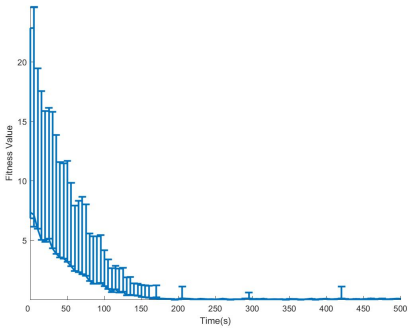


Figure 67: PVM using Spherical Fitness, Distance Noise = 5mm, Messaging Success 100%

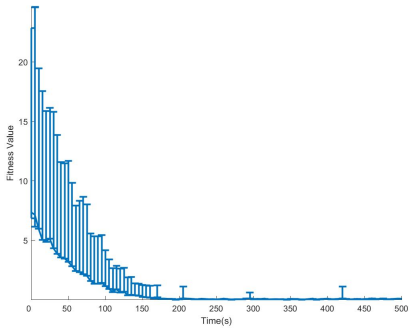


Figure 68: PVM using Spherical Fitness, Distance Noise = 5mm, Messaging Success 100%

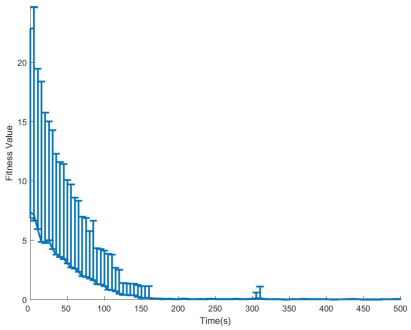


Figure 69: PVM using Spherical Fitness, Distance Noise = 10mm, Messaging Success 100%

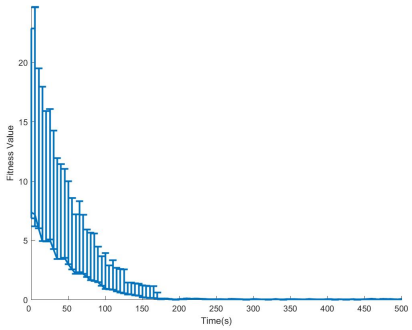


Figure 70: PVM using Spherical Fitness, Distance Noise = 10mm, Messaging Success 80%

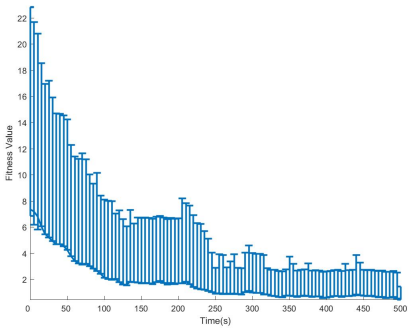


Figure 71: CPVM using Spherical Fitness, Distance Noise = 0mm, Messaging Success 100%

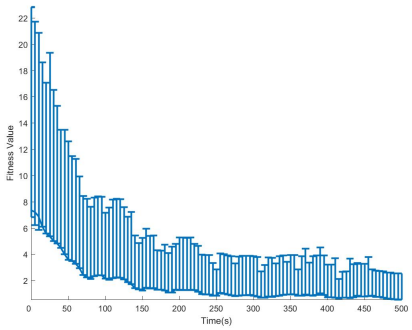


Figure 72: CPVM using Spherical Fitness, Distance Noise = 0mm, Messaging Success 80%

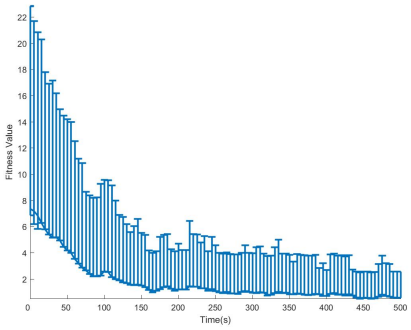


Figure 73: CPVM using Spherical Fitness, Distance Noise = 5mm, Messaging Success 100%

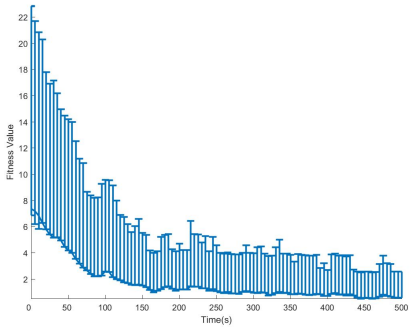


Figure 74: CPVM using Spherical Fitness, Distance Noise = 5mm, Messaging Success 100%

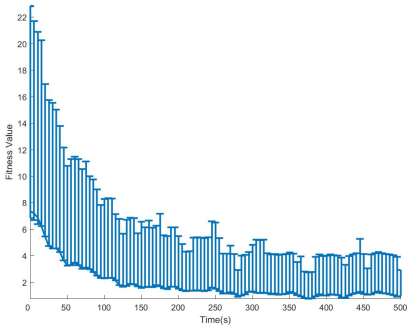


Figure 75: CPVM using Spherical Fitness, Distance Noise = 10mm, Messaging Success 100%

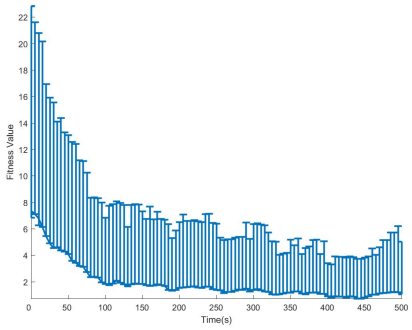


Figure 76: CPVM using Spherical Fitness, Distance Noise = 10mm, Messaging Success 80%

B.3 Social-Cognitive Pseudo Vector Motion (S-CPVM)

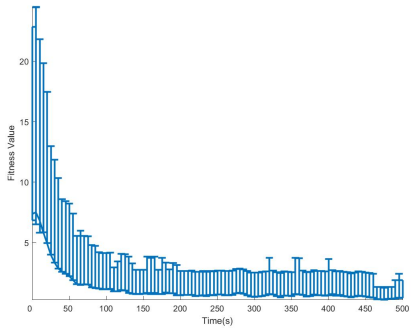


Figure 77: S-CPVM using Spherical Fitness, Distance Noise = 0mm, Messaging Success 100%

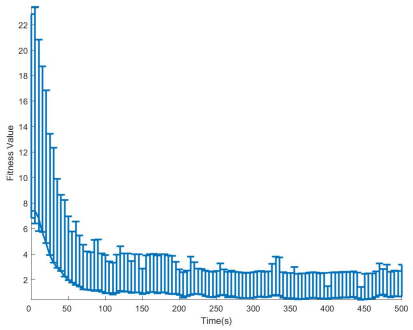


Figure 78: S-CPVM using Spherical Fitness, Distance Noise = 0mm, Messaging Success 80%

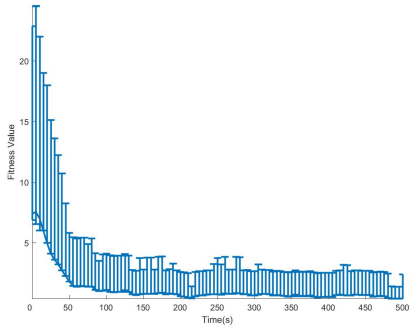


Figure 79: S-CPVM using Spherical Fitness, Distance Noise = 5mm, Messaging Success 100%

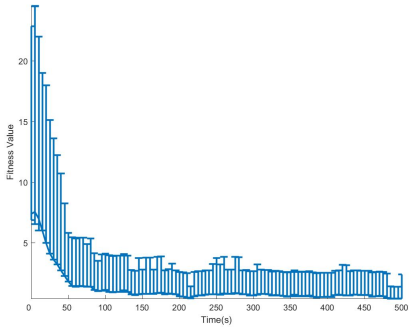


Figure 80: S-CPVM using Spherical Fitness, Distance Noise = 5mm, Messaging Success 80%

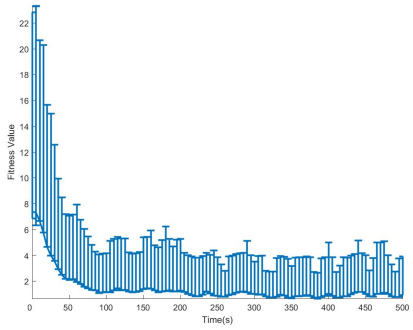


Figure 81: S-CPVM using Spherical Fitness, Distance Noise = 10mm, Messaging Success 100%

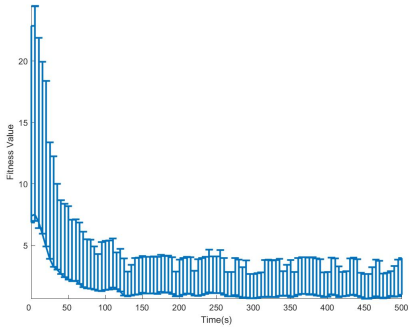


Figure 82: S-CPVM using Spherical Fitness, Distance Noise = 10mm, Messaging Success 80%

B.4 Cumulative Social-Cognitive Pseudo Vector Motion CS-CPVM)

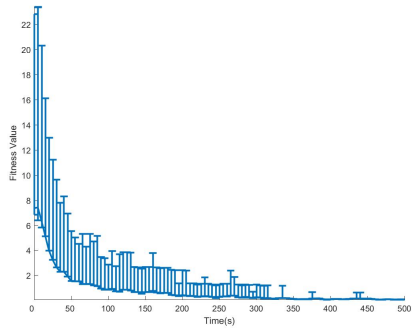


Figure 83: CS-CPVM using Spherical Fitness, Distance Noise = 0mm, Messaging Success 100%

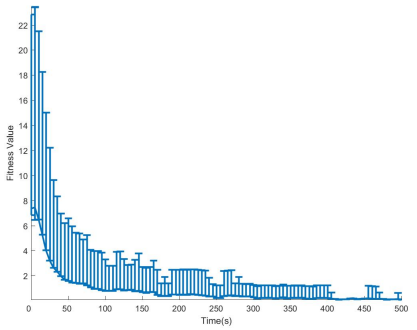


Figure 84: CS-CPVM using Spherical Fitness, Distance Noise = 0mm, Messaging Success 80%

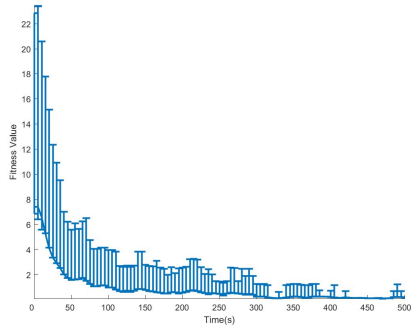


Figure 85: S-CPVM using Spherical Fitness, Distance Noise = 5mm, Messaging Success 100%

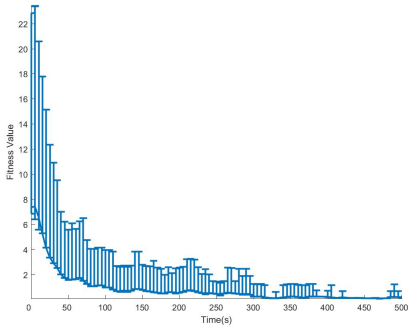


Figure 86: S-CPVM using Spherical Fitness, Distance Noise = 5mm, Messaging Success 100%

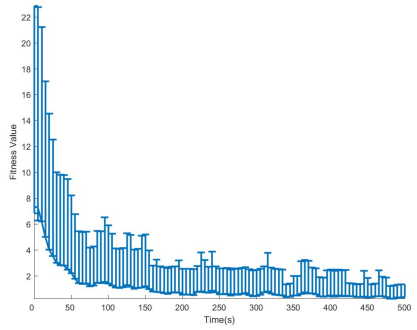


Figure 87: S-CPVM using Spherical Fitness, Distance Noise = 10mm, Messaging Success 100%

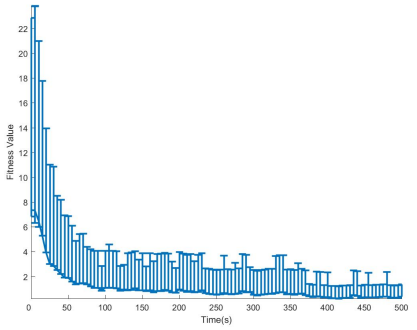


Figure 88: S-CPVM using Spherical Fitness, Distance Noise = 10mm, Messaging Success 80%

B.5 Cumulative Social-Cognitive Pseudo Vector Motion Using Rastrigin Fitness

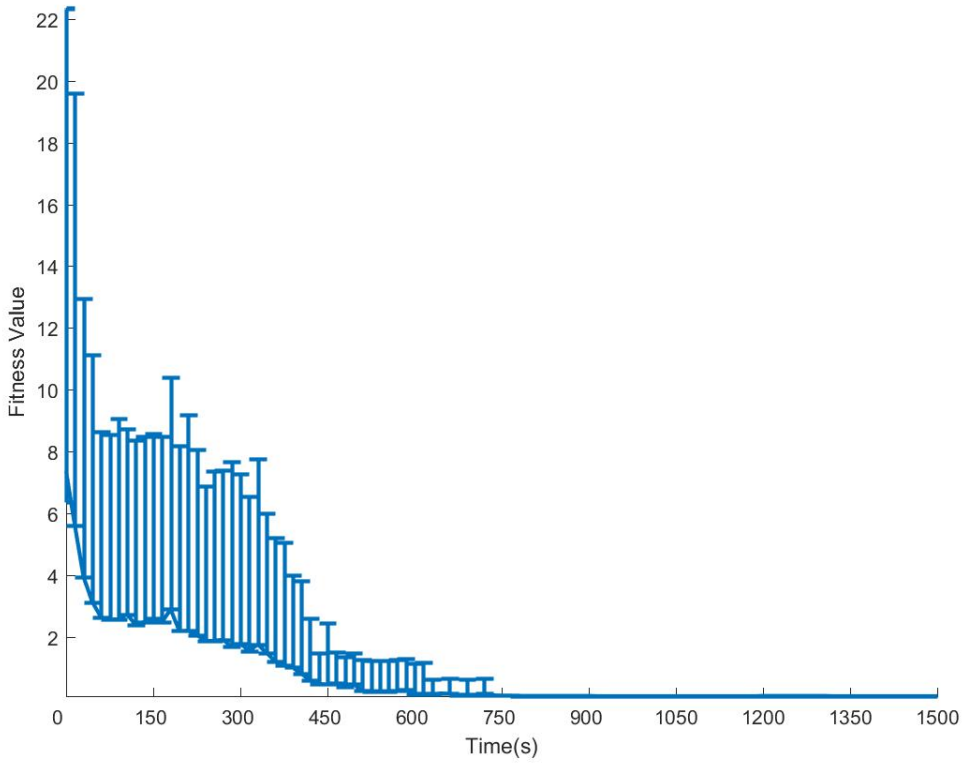


Figure 89: CS-CPVM using Rastrigin Fitness, Distance Noise = 0mm, Messaging Success 100%

APPENDIX C: MATLAB Code

C.1 Fitness Functions

Spherical.m

```
function [output] = Spherical(x)
%SPHERICAL
output = 0;
n = length(x);
for i=1:n
    output = output + x(i).^2;
end
end
```

SphericalNoise.m

```
function [output] = SphericalNoise(x,noise)
%Spherical Function with noise added.
output = 0;
n = length(x);
for i=1:n
    output = output + x(i).^2;
end
output = (sqrt(output) + noise*randn(1))^2;
end
```

SphericalNoiseScaled.m

```
function [output] = SphericalNoiseScaled(x,noise,scaleX,scaleY)
%SPHERICAL
% x = [x,y] : currently for a 2-D search space.
% noise     : noise factor
% scaleX    : scaling factor for X
% scaleY    : scaling factor for Y
n = length(x);
if (n==2)
    x(1) = x(1)/scaleX;
    x(2) = x(2)/scaleY;
end
output = 0;
for i=1:n
    output = output + x(i).^2;
end
output = (sqrt(output) + noise*randn(1))^2;
end
```

Rastrigin.m

```
function [output] = Rastrigin(x)
%RASTRIGIN
n = length(x);
sum = 0;
for i=1:n
    sum = sum + (x(i)^2 - 10*cos(2*pi*x(i)));
end
output = 10*n + sum;
end
```

RastriginNoise.m

```
function [output] = RastriginNoise(x, noise)
%RASTRIGIN
n = length(x);
sum = 0;
for i=1:n
    sum = sum + (x(i)^2 - 10*cos(2*pi*x(i)));
end
output = 10*n + sum;
output = (sqrt(output) + noise*randn(1))^2;
end
```

RastriginNoiseScaled.m

```
function [output] = RastriginNoiseScaled(x, noise, scaleX, scaleY, phase)
%RASTRIGIN
% x = [x,y] : currently for a 2-D search space.
% noise     : noise factor
% scaleX    : scaling factor for X
% scaleY    : scaling factor for Y
% phase     : scaling factor for phase of Rastrigin
n = length(x);
if (n==2)
    x(1) = x(1)/scaleX;
    x(2) = x(2)/scaleY;
end
sum = 0;
for i=1:n
    sum = sum + (x(i)^2 - 10*cos(2*pi*phase*x(i)));
end
output = 10*n + sum;
output = (sqrt(output) + noise*randn(1))^2;
end
```

Rosenbrock.m

```
function [output] = Rosenbrock(x)
%ROSENBROCK
output = 0;
n = length(x);

for i=1:(n-1)
    output = output + 100*(x(i+1) - x(i)^2)^2 + (x(i)-1)^2;
end
end
```

RosenbrockNoise.m

```
function [output] = RosenbrockNoise(x, noise)
%ROSENBROCK
output = 0;
n = length(x);

for i=1:(n-1)
    output = output + 100*(x(i+1) - x(i)^2)^2 + (x(i)-1)^2;
end
output = (sqrt(output) + noise*randn(1))^2;
end
```

RosenbrockNoiseScaled.m

```
function [output] = RosenbrockNoiseScaled(x, noise, scaleX, scaleY)
%ROSENBROCK
% x = [x,y] : currently for a 2-D search space.
% noise     : noise factor
% scaleX    : scaling factor for X
% scaleY    : scaling factor for Y
n = length(x);
if (n==2)
    x(1) = x(1)/scaleX;
    x(2) = x(2)/scaleY;
end
output = 0;
for i=1:(n-1)
    output = output + 100*(x(i+1) - x(i)^2)^2 + (x(i)-1)^2;
end
output = (sqrt(output) + noise*randn(1))^2;
end
```

C.2 Particle Swarm Optimization

BPSO.m

```
function [G, N, pBest, posArray] = BPSO(particles, dimension, ...
    spaceLowX, spaceHighX, spaceLowY, spaceHighY, inertia, posWeight, ...
    gloWeight, benchmark, maxIterations)

%BPSO Basic Particle Swarm Optimization


position = initialize(particles, dimension, spaceLowX, spaceHighX, ...
                      spaceLowY, spaceHighY);
pBest = position;
[gBest,gSum] = checkGlobal(pBest(1,:), pBest, benchmark);

velocity = initialize(particles, dimension, ...
           -1*abs(spaceHighX-spaceLowX),abs(spaceHighX-spaceLowX), ...
           -1*abs(spaceHighY-spaceLowY),abs(spaceHighY-spaceLowY));

%G = zeros(iterations, dimension);
G(1,:) = gBest; %gBest position array.

%N = zeros(iterations,1);
N(1,:) = gSum;  %value of gBest positon array.

posArray = [position];


%Perform the PSO a specified number of iterations.
counter = 0;
while counter < maxIterations - 1
    counter = counter + 1;
    %Update the velocity vectors of the particles.
    velocity = updateVelocity(velocity,inertia,position, ...
                              pBest,gBest,posWeight,gloWeight);
    %Update the particles new position.
    position = updatePosition(position,velocity, ...
                              spaceLowX,spaceHighX,spaceLowY,spaceHighY);
    %Check the particles position against its position of best fitness.
    pBest = checkPosition(pBest,position, benchmark);
    %Check the new pBest against the gBest and store the best fitness.
    [gBest, gSum] = checkGlobal(gBest,pBest, benchmark);
    %Add the new gBest to the G matrix for plotting position.
    G(counter+1,:) = gBest;
    %Save the next number the position returns from the benchmark problem.
    N(counter+1,:) = gSum;
    %Save the positions of all particles.
    posArray = [posArray, position];
end

end
```

NPSO.m

```
function [gBest, nBest, pBest, posArray] = NPSO(particles, dimension, ...
    spaceLowX, spaceHighX, spaceLowY, spaceHighY, inertia, posWeight, ...
    gloWeight, benchmark, maxIterations, nDistance)
%NPSO Particle Swarm Optimization Using Neighborhood Best

position = initialize(particles, dimension, ...
    spaceLowX, spaceHighX, spaceLowY, spaceHighY);
pBest = position;
nBest = pBest;
[gBest(1,:), nBest] = checkNeighbors(0, nBest, position,...
                                     benchmark, nDistance);

velocity = initialize(particles, dimension, ...
               -1*abs(spaceHighX-spaceLowX),abs(spaceHighX-spaceLowX), ...
               -1*abs(spaceHighY-spaceLowY),abs(spaceHighY-spaceLowY));

posArray = [position];

%Perform the PSO a specified number of iterations.
counter = 0;
while counter < maxIterations - 1
    counter = counter + 1;
    %Update the velocity vectors of the particles.
    velocity = updateVelocityN(velocity,inertia,position,pBest,nBest,...
                                posWeight,gloWeight);
    %Update the particles new position.
    position = updatePosition(position,velocity,spaceLowX,spaceHighX,...
                              spaceLowY,spaceHighY);
    %Check the particles position against its position of best fitness.
    %pBest = checkPosition(pBest, position, benchmark);
    [nBest, pBest] = checkPositionN(nBest, pBest, position, benchmark);
    %
    [gBest(counter+1,:), nBest] = checkNeighbors(counter, nBest,...
                                  position, benchmark, nDistance);
    posArray = [posArray, position];
end

end
```

initialize.m

```
function [position] = initialize (length, dimension, ...
                                        spaceLowX, ...
                                           spaceHighX,spaceLowY, ...
                                           spaceHighY)

if (dimension == 2)
    positionX = (spaceLowX + (spaceHighX - spaceLowX) .* rand(length, 1));
    positionY = (spaceLowY + (spaceHighY - spaceLowY) .* rand(length, 1));
else
    fprintf('Dimension error in initialize.');
```

```
end
position = [positionX, positionY];
```

checkPosition.m

```
function [bestPosition] = checkPosition(best, position, benchmark)

[m,n] = size(position);

for i=1:m
    if benchmark(position(i,:)) < benchmark(best(i,:))
        best(i,:) = position(i,:);
    end
end

bestPosition = best;
```

checkPositionN.m

```
function [nBest, pBest] = checkPositionN(nBest, pBest, position, benchmark)

[m,n] = size(position);

for i=1:m
    if benchmark(position(i,:)) < benchmark(pBest(i,:))
        pBest(i,:) = position(i,:);
    end
    if benchmark(position(i,:)) < benchmark(nBest(i,:))
        nBest(i,:) = position(i,:);
    end
end
```

updatePosition.m

```
function [newPosition] = updatePosition (position, velocity, ...
                                         spaceLowX, spaceHighX, ...
                                         spaceLowY, spaceHighY)

[m,n] = size(position);

for i=1:m
    for j=1:n
        position(i,j) = position(i,j) + velocity(i,j);
        if position(i,1) < spaceLowX
            position(i,1) = spaceLowX; %Maybe do something about velocity.
        elseif position(i,1) > spaceHighX %Research this.
            position(i,1) = spaceHighX; %Might increase performance.
        end
        if position(i,2) < spaceLowY
            position(i,2) = spaceLowY; %Maybe do something about velocity.
        elseif position(i,2) > spaceHighY %Research this.
            position(i,2) = spaceHighY; %Might increase performance.
```

```
        end
    end
end

newPosition = position;
```

updateVelocity.m

```
function [newVelocity] = updateVelocity (velocity, velocityWeight, ...
   position, best, globalBest, accelOne, accelTwo)

[m,n] = size(velocity);

for i=1:m
    for j=1:n
        velocity(i,j) = velocityWeight * velocity(i,j) + (accelOne * ...
            rand * (best(i,j) - position(i,j))) + (accelTwo * rand * ...
            (globalBest(1,j) - position(i,j)));
    end
end

newVelocity = velocity;
```

updateVelocityN.m

```
function [newVelocity] = updateVelocityN (velocity, velocityWeight, ...
                            position, pBest, nBest, accelOne, accelTwo)

[m,n] = size(velocity);

for i=1:m
    for j=1:n
        velocity(i,j) = velocityWeight * velocity(i,j) + ...
            (accelOne * rand * (pBest(i,j) - position(i,j))) + ...
            (accelTwo * rand * (nBest(i,j) - position(i,j)));
    end
end

newVelocity = velocity;
```

checkDistance.m

```
function [distance] = checkDistance(posOne,posTwo)

m = length(posOne);
n = length(posTwo);

squaredSum = 0;

if m==n
    for i=1:m
        squaredSum = squaredSum + (posOne(i)-posTwo(i))^2;
```

```
    end
end
distance = sqrt(squaredSum);
end
```

checkGlobal.m

```
function [globalBest, globalSum] = checkGlobal(g, array, benchmark)

[m,n] = size(array);
globalBest = g;
globalSum = benchmark(g);

for i=1:m
    currentSum = benchmark(array(i,:));

%     for j=1:n
%         currentSum = currentSum + benchmark(array(i,j));
%     end

    if currentSum < globalSum
        globalBest = array(i,:);
        globalSum = currentSum;
    end
end
```

checkNeighbors.m

```
function [gBest, nBest] = checkNeighbors(counter, nBest, position, ...
    benchmark, nDistance)

[m,n] = size(position);

gBest = nBest(1,:);

if (rem(counter,2) == 1)
    for i=1:m
        nCurr = nBest(i,:);
        for j=1:m
            if (checkDistance(position(i,:), position(j,:)) < nDistance)
                if benchmark(nBest(j,:)) < benchmark(nCurr) %% ...
                    POSITION, PBEST OR NBEST HERE?
                    nBest(i,:) = nBest(j,:);
                elseif benchmark(nCurr) < benchmark(nBest(j,:))
                    nBest(j,:) = nBest(i,:);
                end
            end
        end
        if (benchmark(nBest(i,:)) < benchmark(gBest))
            gBest = nBest(i,:);
        end
    end
```

```
elseif (rem(counter,2) == 0)
        for i=m:-1:1
        nCurr = nBest(i,:);
        for j=m:-1:1
            if (checkDistance(position(i,:), position(j,:)) < nDistance)
                if benchmark(nBest(j,:)) < benchmark(nCurr) %% ...
                    POSITION, PBEST OR NBEST HERE?
                    nBest(i,:) = nBest(j,:);
                elseif benchmark(nCurr) < benchmark(nBest(j,:))
                    nBest(j,:) = nBest(i,:);
                end
            end
        end
        if (benchmark(nBest(i,:)) < benchmark(gBest))
            gBest = nBest(i,:);
        end
    end
end
end
```

C.3 Simulation Tests

contents.m

```
clear; clc; close all; format long;

% % The data generation for standard PSO is handled in these three scripts.
% SphericalDataGen;
RastriginDataGen;
RosenbrockDataGen;
% % Each script handles all of the cases for individual benchmark ...
    functions.
% % Inside each script, near the top, are lines of code changing the noise
% % level and save folder pathway for that instances data generation.
% % The Spherical and Rosenbrock functions were tested at noise levels of
% % 0, .1, .25, and .5. The Rastrigin function was tested at noise ...
    levels of
% % 0, .5, 1 and 1.5, due to its distance between local minima.
% % The data for these is saved as NoisyData.mat inside of the Data folder,
% % followed by the folder name of the corresponding benchmark ...
    function, and
% % finally by a folder name describing the noise level.


% % The data generation for neighborhood PSO (NPSO) is handled in these
% % three scripts.
% SphericalDataGenNPSO;
% RastriginDataGenNPSO;
% RosenbrockDataGenNPSO;
% % Each script handles all the cases exactly as before, except the data
% % generated uses NPSO. The data is saved in the same fashion as well,
```

```
% % inside of the NPSOData folder.


% % The data processing for the standard PSO is handled in these three
% % scripts.
% SphericalErrorProc;
% RastriginErrorProc;
% RosenbrockErrorProc;
% % Each script handles all of the cases for individual benchmark ...
    functions.
% % Inside each script, near the top, are lines of code changing which data
% % is loaded for processing. The processed data is then saved in the same
% % folder as where it was retrieved from, and named ProcessedData.mat. ...
    These
% % scripts also generate graphs to represent the data and save them in the
% % same folder as well.

% % The data processing for the neighborhood PSO (NPSO) is handled in these
% % three scripts.
% SphericalErrorProcNPSO;
% RastriginErrorProcNPSO;
% RosenbrockErrorProcNPSO;
% % Each script handles all the cases exactly as before, except the data
% % generated uses NPSO. The data is saved in the same fashion as well,
% % inside of the NPSOData folder.
% %

% % Used for moving the figure files with more meaningful names.
% figureSaves;
```

SphericalDataGen.m

```
clear; clc; close all; format long;

% Does all the PSO data generation for the Spherical function.

% Folder path for saves.
folderPath = 'Data/Spherical/';
saveName = 'NoisyData';

% Parameters that change for data generation.
% This is a noise level and a folder pair to save the data in.
noise = 0; folderName = 'Noise00';
% noise = 0.10; folderName = 'Noise10';
% noise = 0.25; folderName = 'Noise25';
% noise = 0.50; folderName = 'Noise50';

% Benchmark function is the Spherical function.
benchmark = @(x) SphericalNoise(x,noise);

% Particle Position Parameters for PSO
particles  = 30; %Number of particles in the swarm.
dimension  =  2; %Dimension of the search space.
```

```
spaceLow   = -5.12; %Minimum value of the search space.
spaceHigh  =  5.12; %Maximum value of the search space.

% Particle Velocity Parameters for PSO
inertia    =  .5; %Particles own movement weight.
posWeight  = 2; %Particles own best position influence.
gloWeight  = 1; %Global best position influence.

maxIterations = 100; % PSO counter cannot excede this.
MCIterations = 100;

pCell = cell(MCIterations, 1);
gCell = cell(MCIterations, 1);

MCbest = zeros(MCIterations,2);

MCprog = zeros(MCIterations, maxIterations);

for i=1:MCIterations
    [G, N, pBest, posArray] = ...
       BPSO(particles,dimension,spaceLow,spaceHigh,...
                      inertia, posWeight, gloWeight,...
                      benchmark, maxIterations);
    MCbest(i,:) = G(end,:);
    for j=1:length(G)
        MCprog(i,j) = Spherical(G(j,:));
    end
    pCell{i} = pBest;
    gCell{i} = G;
end

savePath = [folderPath folderName];
if (exist(savePath,'dir') == 0)
    mkdir(savePath);
end

saveName = [savePath '/' saveName];

save(saveName, 'particles', 'dimension', 'spaceLow', 'spaceHigh', ...
                'inertia', 'posWeight', 'gloWeight', 'MCbest', ...
                'MCprog', 'pCell', 'gCell', 'noise', 'savePath',...
                'maxIterations');
```

RastriginDataGen.m

```
clear; clc; close all; format long;

% Does all the data generation for the Rastrigin function.

% Folder path for saves.
folderPath = 'Data/Rastrigin/';
saveName = 'NoisyData';

```

```
% Parameters that change for data generation.
% This is a noise level and a folder pair to save the data in.
noise = 0; folderName = 'Noise00';
% noise = 0.50; folderName = 'Noise50';
% noise = 1.00; folderName = 'Noise100';
% noise = 1.00; folderName = 'Noise100_I2';
% noise = 1.50; folderName = 'Noise150';

% Benchmark function is the Rastrigin function.
benchmark = @(x) RastriginNoise(x,noise);

% Particle Position Parameters for PSO
particles  = 30; %Number of particles in the swarm.
dimension  =  2; %Dimension of the search space.
spaceLow   = -5.12; %Minimum value of the search space.
spaceHigh  =  5.12; %Maximum value of the search space.

% Particle Velocity Parameters for PSO
inertia    =  .5; %Particles own movement weight.
posWeight  = 2; %Particles own best position influence.
gloWeight  = 1; %Global best position influence.

maxIterations = 100; % PSO counter cannot excede this.
MCIterations = 100;

pCell = cell(MCIterations, 1);
gCell = cell(MCIterations, 1);

MCbest = zeros(MCIterations,2);

MCprog = zeros(MCIterations, maxIterations);

for i=1:MCIterations
    [G, N, pBest] = BPSO(particles,dimension,spaceLow,spaceHigh,...
                         inertia, posWeight, gloWeight,...
                         benchmark, maxIterations);
    MCbest(i,:) = G(end,:);
    for j=1:length(G)
        MCprog(i,j) = Rastrigin(G(j,:));
    end
    pCell{i} = pBest;
    gCell{i} = G;
end

savePath = [folderPath folderName];
if (exist(savePath,'dir') == 0)
    mkdir(savePath);
end

saveName = [savePath '/' saveName];

save(saveName, 'particles', 'dimension', 'spaceLow', 'spaceHigh', ...
                'inertia', 'posWeight', 'gloWeight', 'MCbest', ...
                'MCprog', 'pCell', 'gCell', 'noise', 'savePath',...
```

```
63                  'maxIterations');
```

RosenbrockDataGen.m

```
1  clear; clc; close all; format long;
2
3  % Does all the data generation for the Rosenbrock function.
4
5  % Folder path for saves.
6  folderPath = 'Data/Rosenbrock/';
7  saveName = 'NoisyData';
8
9  % Parameters that change for data generation.
10 % This is a noise level and a folder pair to save the data in.
11 noise = 0; folderName = 'Noise00';
12 % noise = 0.10; folderName = 'Noise10';
13 % noise = 0.25; folderName = 'Noise25';
14 % noise = 0.50; folderName = 'Noise50';
15
16 % Benchmark function is the Rosenbrock function.
17 benchmark = @(x) RosenbrockNoise(x,noise);
18
19 % Particle Position Parameters for PSO
20 particles  = 30; %Number of particles in the swarm.
21 dimension  =  2; %Dimension of the search space.
22 spaceLow   = -2.048; %Minimum value of the search space.
23 spaceHigh  =  2.048; %Maximum value of the search space.
24
25 % Particle Velocity Parameters for PSO
26 inertia    =  .5; %Particles own movement weight.
27 posWeight  = 2; %Particles own best position influence.
28 gloWeight  = 1; %Global best position influence.
29
30 maxIterations = 100; % PSO counter cannot excede this.
31 MCIterations = 100;
32
33 pCell = cell(MCIterations, 1);
34 gCell = cell(MCIterations, 1);
35
36 MCbest = zeros(MCIterations,2);
37
38 MCprog = zeros(MCIterations, maxIterations);
39
40 for i=1:MCIterations
41     [G, N, pBest] = BPSO(particles,dimension,spaceLow,spaceHigh,...
42                          inertia, posWeight, gloWeight,...
43                          benchmark, maxIterations);
44     MCbest(i,:) = G(end,:);
45     for j=1:length(G)
46         MCprog(i,j) = Rosenbrock(G(j,:));
47     end
48     pCell{i} = pBest;
49     gCell{i} = G;
```

```
end

savePath = [folderPath folderName];
if (exist(savePath,'dir') == 0)
    mkdir(savePath);
end

saveName = [savePath '/' saveName];

save(saveName, 'particles', 'dimension', 'spaceLow', 'spaceHigh', ...
                'inertia', 'posWeight', 'gloWeight', 'MCbest', ...
                'MCprog', 'pCell', 'gCell', 'noise', 'savePath',...
                'maxIterations');
```

SphericalDataGenNPSO.m

```
clear; clc; close all; format long;

% Does all the NPSO data generation for the Spherical function.

% Folder path for saves.
folderPath = 'NPSOData/Spherical/';
saveName = 'NoisyData';

% Parameters that change for data generation.
% This is a noise level and a folder pair to save the data in.
noise = 0; folderName = 'Noise00';
% noise = 0.10; folderName = 'Noise10';
% noise = 0.25; folderName = 'Noise25';
% noise = 0.50; folderName = 'Noise50';

% Benchmark function is the Spherical function.
benchmark = @(x) SphericalNoise(x,noise);

% Particle Position Parameters for PSO
particles  = 30; %Number of particles in the swarm.
dimension  =  2; %Dimension of the search space.
spaceLow   = -5.12; %Minimum value of the search space.
spaceHigh  =  5.12; %Maximum value of the search space.

% Particle Velocity Parameters for PSO
inertia    =  .5; %Particles own movement weight.
posWeight  = 2; %Particles own best position influence.
neiWeight  = 1; %Global best position influence.

maxIterations = 100; % PSO counter cannot excede this.
nDistance = 1; % Distance value for neighborhood calculations.
MCIterations = 100;

MCbest = zeros(MCIterations,2);
MCprog = zeros(MCIterations, maxIterations);

gCell = cell(MCIterations, 1);
```

```
38
39
40 for i=1:MCIterations
41     [g,n,p] = NPSO(particles,dimension,spaceLow,spaceHigh,...
42               inertia,posWeight,neiWeight,...
43               benchmark, maxIterations, nDistance);
44     MCbest(i,:) = g(end,:);
45     for j=1:length(g)
46         MCprog(i,j) = Spherical(g(j,:));
47     end
48     gCell{i} = g;
49 end
50
51 savePath = [folderPath folderName];
52 if (exist(savePath,'dir') == 0)
53     mkdir(savePath);
54 end
55
56 saveName = [savePath '/' saveName];
57
58 save(saveName, 'particles', 'dimension', 'spaceLow', 'spaceHigh',...
59                'inertia', 'posWeight', 'neiWeight', 'p', 'n',...
60                'MCbest', 'MCprog', 'gCell', 'noise', 'savePath',...
61                'maxIterations');
```

RastriginDataGenNPSO.m

```
1  clear; clc; close all; format long;
2
3  % Does all the NPSO data generation for the Rastrigin function.
4
5  % Folder path for saves.
6  folderPath = 'NPSOData/Rastrigin/';
7  saveName = 'NoisyData';
8
9  % Parameters that change for data generation.
10 % This is a noise level and a folder pair to save the data in.
11 noise = 0; folderName = 'Noise00';
12 % noise = 0.50; folderName = 'Noise50';
13 % noise = 1.00; folderName = 'Noise100';
14 % noise = 1.00; folderName = 'Noise100_I2';
15 % noise = 1.50; folderName = 'Noise150';
16
17 % Benchmark function is the Rastrigin function.
18 benchmark = @(x) RastriginNoise(x,noise);
19
20 % Particle Position Parameters for PSO
21 particles  = 30; %Number of particles in the swarm.
22 dimension  =  2; %Dimension of the search space.
23 spaceLow   = -5.12; %Minimum value of the search space.
24 spaceHigh  =  5.12; %Maximum value of the search space.
25
26 % Particle Velocity Parameters for PSO
```

```
inertia    =  .5; %Particles own movement weight.
posWeight  = 2; %Particles own best position influence.
neiWeight  = 1; %Global best position influence.

maxIterations = 100; % PSO counter cannot excede this.
nDistance = 1; % Distance value for neighborhood calculations.
MCIterations = 100;

MCbest = zeros(MCIterations,2);
MCprog = zeros(MCIterations, maxIterations);

gCell = cell(MCIterations, 1);


for i=1:MCIterations
    [g,n,p] = NPSO(particles,dimension,spaceLow,spaceHigh,...
              inertia,posWeight,neiWeight,...
              benchmark, maxIterations, nDistance);
    MCbest(i,:) = g(end,:);
    for j=1:length(g)
        MCprog(i,j) = Rastrigin(g(j,:));
    end
    gCell{i} = g;
end

savePath = [folderPath folderName];
if (exist(savePath,'dir') == 0)
    mkdir(savePath);
end

saveName = [savePath '/' saveName];

save(saveName, 'particles', 'dimension', 'spaceLow', 'spaceHigh',...
               'inertia', 'posWeight', 'neiWeight', 'p', 'n',...
               'MCbest', 'MCprog', 'gCell', 'noise', 'savePath',...
               'maxIterations');
```

RosenbrockDataGenNPSO.m

```
clear; clc; close all; format long;

% Does all the NPSO data generation for the Rosenbrock function.

% Folder path for saves.
folderPath = 'NPSOData/Rosenbrock/';
saveName = 'NoisyData';

% Parameters that change for data generation.
% This is a noise level and a folder pair to save the data in.
noise = 0; folderName = 'Noise00';
% noise = 0.10; folderName = 'Noise10';
% noise = 0.25; folderName = 'Noise25';
% noise = 0.50; folderName = 'Noise50';
```

```
15
16 % Benchmark function is the Rosenbrock function.
17 benchmark = @(x) RosenbrockNoise(x,noise);
18
19 % Particle Position Parameters for PSO
20 particles  = 30; %Number of particles in the swarm.
21 dimension  =  2; %Dimension of the search space.
22 spaceLow   = -2.048; %Minimum value of the search space.
23 spaceHigh  =  2.048; %Maximum value of the search space.
24
25 % Particle Velocity Parameters for PSO
26 inertia    =  .5; %Particles own movement weight.
27 posWeight  = 2; %Particles own best position influence.
28 neiWeight  = 1; %Global best position influence.
29
30 maxIterations = 100; % PSO counter cannot excede this.
31 nDistance = 1; % Distance value for neighborhood calculations.
32 MCIterations = 100;
33
34 MCbest = zeros(MCIterations,2);
35 MCprog = zeros(MCIterations, maxIterations);
36
37 gCell = cell(MCIterations, 1);
38
39
40 for i=1:MCIterations
41     [g,n,p] = NPSO(particles,dimension,spaceLow,spaceHigh,...
42               inertia,posWeight,neiWeight,...
43               benchmark, maxIterations, nDistance);
44     MCbest(i,:) = g(end,:);
45     for j=1:length(g)
46         MCprog(i,j) = Rosenbrock(g(j,:));
47     end
48     gCell{i} = g;
49 end
50
51 savePath = [folderPath folderName];
52 if (exist(savePath,'dir') == 0)
53     mkdir(savePath);
54 end
55
56 saveName = [savePath '/' saveName];
57
58 save(saveName, 'particles', 'dimension', 'spaceLow', 'spaceHigh',...
59                'inertia', 'posWeight', 'neiWeight', 'p', 'n',...
60                'MCbest', 'MCprog', 'gCell', 'noise', 'savePath',...
61                'maxIterations');
```

SphericalErrorProc.m

```
clear; clc; close all; format long;

% Processes noisy data from the PSO that used the Spherical function.

% Potential load paths for different noise levels.
% load('Data/Spherical/Noise00/NoisyData.mat');
% load('Data/Spherical/Noise10/NoisyData.mat');
% load('Data/Spherical/Noise25/NoisyData.mat');
% load('Data/Spherical/Noise50/NoisyData.mat');
% load('ScaledData/Spherical/Noise00/NoisyData.mat');
% load('ScaledData/Spherical/Noise10/NoisyData.mat');
% load('ScaledData/Spherical/Noise25/NoisyData.mat');
% load('ScaledData/Spherical/Noise50/NoisyData.mat');
load('ScaledLDIPSOData/Spherical/Noise00/NoisyData.mat');
% load('ScaledLDIPSOData/Spherical/Noise25/NoisyData.mat');

% benchmark = @(x) Spherical(x);
benchmark = @(x) SphericalNoiseScaled(x,0,scaleX,scaleY);

% Save name for processed data.
saveName = 'ProcessedData';

actual = 0;
epsilon = .01;
totalError = 0;
success = 0;
seed = 12;

rng(seed);   % Makes data choice repeatable.

[x,y] = meshgrid(spaceLowX:1:spaceHighX,spaceLowY:1:spaceHighY);
[m,n] = size(x);
z = zeros(m,n);
for i=1:m
    for j=1:n
        z(i,j) = benchmark([x(i,j),y(i,j)]);
    end
end

% figure(1); clf(1); hold on;
% %axis equal;
% title(['Final gBest Value']);
% contour(x,y,z);

%Plot a circle.
% r = sqrt(epsilon);
% ang=0:0.01:2*pi;
% xp= r*cos(ang);
% yp= r*sin(ang);
% plot(0+xp,0+yp,'red');
```

```
51
52 error = zeros(1,length(MCbest));
53 for i=1:length(MCbest)
54     error(i) = abs(benchmark(MCbest(i,:)) - actual);
55     if  error < epsilon
56 %         plot(MCbest(i,1),MCbest(i,2),'red*');
57         success = success + 1;
58     else
59 %         plot(MCbest(i,1),MCbest(i,2),'black*');
60     end
61     histData(i) = sqrt(MCbest(i,1)^2 + MCbest(i,2)^2);
62 end
63
64 maxError = max(error);
65 minError = min(error);
66 totalError = sum(error);
67 averageError = totalError/i;
68 successPercent = success/i;
69
70 consistency = sqrt(sum((error-averageError).^2));
71
72 saveName = [savePath '/' saveName];
73 % save(saveName, 'epsilon', 'averageError', 'successPercent', 'seed');
74
75 % figName = [savePath '/' 'GBestAccuracyFig'];
76 % saveas(1, [figName '.fig']);
77 % saveas(1, [figName '.jpg']);
78 % saveas(1, [figName '.pdf']);
79
80 index = randi(100,1);
81
82 %%%%%%% Figure 2 %%%%%% Plots a pBest progression. %%%%%%%
83 % pBest = cell2mat(pCell(index));
84 % figure(2); clf(2); hold on;
85 %
86 % Plot a circle.
87 % r = sqrt(epsilon);
88 % ang=0:0.01:2*pi;
89 % xp= r*cos(ang);
90 % yp= r*sin(ang);
91 % plot(0+xp,0+yp,'red');
92 %
93 % title(['Final pBest location for run' num2str(index)...
94 %        '. noise = ' num2str(noise)]);
95 % for i=1:length(pBest)
96 %     plot(pBest(i,1),pBest(i,2), 'black*');
97 % end
98 %
99 % figName = [savePath '/' 'PbestAccuracyFig'];
100 % saveas(2, [figName '.fig']);
101 % saveas(2, [figName '.jpg']);
102 % saveas(2, [figName '.pdf']);
103
104 % figure(2); clf(2); hold on;
```

```
105 % contour(x,y,z);
106 % posMat = cell2mat(xCell(1));
107 % onePart = posMat(1,:);
108 % % x = posMat(:,1);
109 % % y = posMat(:,2);
110 % % plot(x,y);
111 % u = [];
112 % v = [];
113 % for i=1:maxIterations
114 %     u = [u; onePart(1,2*i-1)];
115 %     v = [v; onePart(1,2*i)];
116 % end
117 % plot(u,v);
118 % % for i=1:length(maxIterations)
119 % %     posMat = cell2mat(xCell(i));
120 % % end
121
122
123
124 % figure(3); clf(3); hold on;
125 % %title('gBest progression for all 100 MC runs');
126 % xlabel('Iteration Number');
127 % ylabel('Function Value');
128 %
129 % for i=1:length(gCell)
130 %     gBest = cell2mat(gCell(i));
131 %     gProg = zeros(length(gBest),1);
132 %     for j=1:length(gBest)
133 %         gProg(j) = Rastrigin(gBest(j,:));
134 %     end
135 %     plot(gProg);
136 % end
137 %
138 % figName = [savePath '/' 'GbestProgressionFig'];
139 % saveas(3, [figName '.fig']);
140 % saveas(3, [figName '.jpg']);
141 % saveas(3, [figName '.pdf']);
142
143
144 length = size(MCprog,2);
145 avg = zeros(1,length);
146 stddev = zeros(1, length);
147 for i=1:length
148     avg(i) = mean(MCprog(:,i));
149     stddev(i) = std(MCprog(:,i));
150 end
151
152 pTile = prctile(MCprog,[10,90],1);
153 lower = pTile(1,:);
154 upper = pTile(2,:);
155 maxIterations = 100;
156 X = 0:maxIterations-1;
157
158
```

```
% figure(4); clf(4); hold on;
% title('Distribution of the Fitness Value of the Global Best (1000 ...
    runs).');
% xlabel('Iteration Number');
% ylabel('Average Distribution');
% axis([min(X),max(X),-inf,inf]);  % Set axis.
%
% set(gca,'fontsize',18);  % Change the font size.
% title('Fitness of PSO');
%
% errorbar(avg,stddev);
% errorbar(X,avg,lower,upper);
%
% figName = [savePath '/' 'ErrorBarFig'];
% saveas(4, [figName '.fig']);
% saveas(4, [figName '.jpg']);
% saveas(4, [figName '.pdf']);


% figure(5); clf(5); hold on;
% hist(histData, 20);
% hist(histData, [0:0.05:1.2]);
% hist(histData, [0:0.07:2]);
% xlabel('Distance from global optimum.');
% ylabel('Number in bin. (/100)');
%
% axis([0,1.2,0,20]);
% xlim([0,1.2]);
% xlim([0,2]);
% set(gca,'fontsize',18);  % Change the font size.
% title('PSO Distance from Optimum');
%
% figName = [savePath '/' 'HistDistFig'];
% saveas(5, [figName '.fig']);
% saveas(5, [figName '.jpg']);
% saveas(5, [figName '.pdf']);

% figure(6); clf(6); hold on;
% hist(MCprog(:,end), 20);
% xlabel('Fitness value.');
% ylabel('Number in bin. (/100)');
%
% figName = [savePath '/' 'HistFitFig'];
% saveas(6, [figName '.fig']);
% saveas(6, [figName '.jpg']);
% saveas(6, [figName '.pdf']);

figure(7); clf(7); hold on;
xlabel('Iteration Number','FontSize', 20);
ylabel('Average Distribution','FontSize', 20);
title('Fitness of PSO','FontSize', 20);
axis([min(X),max(X),-inf,inf]);
errorbar(X,avg,lower,upper,'LineWidth',2);
axes('Position',[.5,.5,.4,.4]);
```

```
xbin = linspace(0,2.5*10^(-9),20);
% xbin = linspace(0,25,20);
hist(histData, xbin);
axis([0,2.5*10^(-9),-inf,inf]);
% axis([0,25,-inf,inf]);
xlabel('Distance from global optimum.', 'FontSize', 14);
ylabel('Number in bin. (/100)', 'FontSize', 14);

figName = [savePath '/' 'ErrorHistFig'];
saveas(7, [figName '.fig']);
saveas(7, [figName '.jpg']);
saveas(7, [figName '.pdf']);
```

RastriginErrorProc.m

```
clear; clc; close all; format long;

% Processes noisy data from the PSO that used the Rastrigin function.

% Potential load paths for different noise levels.
% load('Data/Rastrigin/Noise00/NoisyData.mat');
% load('Data/Rastrigin/Noise50/NoisyData.mat');
% load('Data/Rastrigin/Noise100/NoisyData.mat');
% load('Data/Rastrigin/Noise100_I2/NoisyData.mat');
% load('Data/Rastrigin/Noise150/NoisyData.mat');
load('ScaledData/Rastrigin/Noise00/NoisyData.mat');
% load('ScaledData/Rastrigin/Noise50/NoisyData.mat');
% load('ScaledData/Rastrigin/Noise100/NoisyData.mat');
% load('ScaledData/Rastrigin/Noise150/NoisyData.mat');
% load('ScaledData/Rastrigin/Noise200/NoisyData.mat');
% load('ScaledData/Rastrigin/Noise400/NoisyData.mat');
% load('ScaledData/Rastrigin/Noise1500/NoisyData.mat');

% benchmark = @(x) Rastrigin(x);
benchmark = @(x) RastriginNoiseScaled(x,0,scaleX,scaleY,.25);

% Save name for processed data.
saveName = 'ProcessedData';

actual = 0;
epsilon = .01;
totalError = 0;
success = 0;
seed = 12;

rng(seed);   % Makes data choice repeatable.

% [x,y] = meshgrid(spaceLowX:.1:spaceHighX,spaceLowY:.1:spaceHighY);
[x,y] = meshgrid(spaceLowX:1:spaceHighX,spaceLowY:1:spaceHighY);
[m,n] = size(x);
z = zeros(m,n);
for i=1:m
    for j=1:n
```

```
        z(i,j) = benchmark([x(i,j),y(i,j)]);
    end
end

% figure(1); clf(1); hold on;
% axis equal;
% title(['Final gBest Value']);
% contour(x,y,z);

%Plot a circle.
% r = sqrt(epsilon);
% ang=0:0.01:2*pi;
% xp= r*cos(ang);
% yp= r*sin(ang);
% plot(0+xp,0+yp,'red');

error = zeros(1,length(MCbest));
for i=1:length(MCbest)
    error(i) = abs(benchmark(MCbest(i,:)) - actual);
    if  error < epsilon
%         plot(MCbest(i,1),MCbest(i,2),'red*');
        success = success + 1;
    else
%         plot(MCbest(i,1),MCbest(i,2),'black*');
    end
    histData(i) = sqrt(MCbest(i,1)^2 + MCbest(i,2)^2);
end

maxError = max(error);
minError = min(error);
totalError = sum(error);
averageError = totalError/i;
successPercent = success/i;

consistency = sqrt(sum((error-averageError).^2));

saveName = [savePath '/' saveName];
% save(saveName, 'epsilon', 'averageError', 'successPercent', 'seed');

% figName = [savePath '/' 'GBestAccuracyFig'];
% saveas(1, [figName '.fig']);
% saveas(1, [figName '.jpg']);
% saveas(1, [figName '.pdf']);

index = randi(100,1);

%%%%%%% Figure 2 %%%%%% Plots a pBest progression. %%%%%%%
% pBest = cell2mat(pCell(index));
% figure(2); clf(2); hold on;
%
% Plot a circle.
% r = sqrt(epsilon);
% ang=0:0.01:2*pi;
% xp= r*cos(ang);
```

```
% yp= r*sin(ang);
% plot(0+xp,0+yp,'red');
%
% title(['Final pBest location for run' num2str(index)...
%        '. noise = ' num2str(noise)]);
% for i=1:length(pBest)
%     plot(pBest(i,1),pBest(i,2), 'black*');
% end
%
% figName = [savePath '/' 'PbestAccuracyFig'];
% saveas(2, [figName '.fig']);
% saveas(2, [figName '.jpg']);
% saveas(2, [figName '.pdf']);



% figure(3); clf(3); hold on;
% %title('gBest progression for all 100 MC runs');
% xlabel('Iteration Number');
% ylabel('Function Value');
%
% for i=1:length(gCell)
%     gBest = cell2mat(gCell(i));
%     gProg = zeros(length(gBest),1);
%     for j=1:length(gBest)
%         gProg(j) = Rastrigin(gBest(j,:));
%     end
%     plot(gProg);
% end
%
% figName = [savePath '/' 'GbestProgressionFig'];
% saveas(3, [figName '.fig']);
% saveas(3, [figName '.jpg']);
% saveas(3, [figName '.pdf']);


length = size(MCprog,2);
avg = zeros(1,length);
stddev = zeros(1, length);
for i=1:length
    avg(i) = mean(MCprog(:,i));
    stddev(i) = std(MCprog(:,i));
end

pTile = prctile(MCprog,[10,90],1);
lower = pTile(1,:);
upper = pTile(2,:);
maxIterations = 100;
X = 0:maxIterations-1;


% figure(4); clf(4); hold on;
% title('Distribution of the Fitness Value of the Global Best (1000 ...
    runs).');
```

```
146 % xlabel('Iteration Number');
147 % ylabel('Average Distribution');
148 % axis([min(X),max(X),-inf,inf]);  % Set axis.
149 %
150 % set(gca,'fontsize',18);  % Change the font size.
151 % title('Fitness of PSO');
152 %
153 % errorbar(avg,stddev);
154 % errorbar(X,avg,lower,upper);
155 %
156 % figName = [savePath '/' 'ErrorBarFig'];
157 % saveas(4, [figName '.fig']);
158 % saveas(4, [figName '.jpg']);
159 % saveas(4, [figName '.pdf']);
160
161
162 % figure(5); clf(5); hold on;
163 % hist(histData, 20);
164 % hist(histData, [0:0.05:1.2]);
165 % hist(histData, [0:0.07:2]);
166 % xlabel('Distance from global optimum.');
167 % ylabel('Number in bin. (/100)');
168 %
169 % axis([0,1.2,0,20]);
170 % xlim([0,1.2]);
171 % xlim([0,2]);
172 % set(gca,'fontsize',18);  % Change the font size.
173 % title('PSO Distance from Optimum');
174 %
175 % figName = [savePath '/' 'HistDistFig'];
176 % saveas(5, [figName '.fig']);
177 % saveas(5, [figName '.jpg']);
178 % saveas(5, [figName '.pdf']);
179
180 % figure(6); clf(6); hold on;
181 % hist(MCprog(:,end), 20);
182 % xlabel('Fitness value.');
183 % ylabel('Number in bin. (/100)');
184 %
185 % figName = [savePath '/' 'HistFitFig'];
186 % saveas(6, [figName '.fig']);
187 % saveas(6, [figName '.jpg']);
188 % saveas(6, [figName '.pdf']);
189
190 figure(7); clf(7); hold on;
191 xlabel('Iteration Number','FontSize', 20);
192 ylabel('Average Distribution','FontSize', 20);
193 title('Fitness of PSO','FontSize', 20);
194 axis([min(X),max(X),-inf,inf]);
195 errorbar(X,avg,lower,upper,'LineWidth',2);
196 axes('Position',[.5,.5,.4,.4]);
197 xbin = linspace(0,5*10^(-6),20);
198 % xbin = linspace(0,80,20);
199 % xbin = linspace(0,500,20);
```

```
hist(histData, xbin);
axis([0,5*10^(-6),-inf,inf]);
% axis([0,80,-inf,inf]);
% axis([0,500,-inf,inf]);
xlabel('Distance from global optimum.','FontSize', 14);
ylabel('Number in bin. (/100)','FontSize', 14);

figName = [savePath '/' 'ErrorHistFig'];
saveas(7, [figName '.fig']);
saveas(7, [figName '.jpg']);
saveas(7, [figName '.pdf']);
```

RosenbrockErrorProc.m

```
clear; clc; close all; format long;

% Processes noisy data from the PSO that used the Rosenbrock function.

% Potential load paths for different noise levels.
% load('Data/Rosenbrock/Noise00/NoisyData.mat');
% load('Data/Rosenbrock/Noise10/NoisyData.mat');
% load('Data/Rosenbrock/Noise25/NoisyData.mat');
% load('Data/Rosenbrock/Noise50/NoisyData.mat');
% load('ScaledData/Rosenbrock/Noise00/NoisyData.mat');
% load('ScaledData/Rosenbrock/Noise10/NoisyData.mat');
% load('ScaledData/Rosenbrock/Noise25/NoisyData.mat');
% load('ScaledData/Rosenbrock/Noise50/NoisyData.mat');
% load('ScaledLDIPSOData/Rosenbrock/Noise00/NoisyData.mat');
load('ScaledLDIPSOData/Rosenbrock/Noise25/NoisyData.mat');

% benchmark = @(x) Rosenbrock(x);
benchmark = @(x) RosenbrockNoiseScaled(x,0,scaleX,scaleY);

% Save name for processed data.
saveName = 'ProcessedData';

actual = 0;
epsilon = .01;
totalError = 0;
success = 0;
seed = 12;

rng(seed);   % Makes data choice repeatable.

% [x,y] = meshgrid(spaceLow:.1:spaceHigh,spaceLow:.1:spaceHigh);
[x,y] = meshgrid(spaceLowX:1:spaceHighX,spaceLowY:1:spaceHighY);
[m,n] = size(x);
z = zeros(m,n);
for i=1:m
    for j=1:n
        z(i,j) = benchmark([x(i,j),y(i,j)]);
    end
end
```

```
40
41 % figure(1); clf(1); hold on;
42 % axis equal;
43 % title(['Final gBest Value']);
44 % contour(x,y,z);
45
46 %Plot a circle.
47 % r = sqrt(epsilon);
48 % ang=0:0.01:2*pi;
49 % xp= r*cos(ang);
50 % yp= r*sin(ang);
51 % plot(0+xp,0+yp,'red');
52
53 error = zeros(1,length(MCbest));
54 for i=1:length(MCbest)
55     error(i) = abs(benchmark(MCbest(i,:)) - actual);
56     if  error < epsilon
57 %         plot(MCbest(i,1),MCbest(i,2),'red*');
58         success = success + 1;
59     else
60 %         plot(MCbest(i,1),MCbest(i,2),'black*');
61     end
62 %     histData(i) = sqrt((1 - MCbest(i,1))^2 + (1 - MCbest(i,2))^2);
63     histData(i) = sqrt((200 - MCbest(i,1))^2 + (100 - MCbest(i,2))^2);
64 end
65
66 maxError = max(error);
67 minError = min(error);
68 totalError = sum(error);
69 averageError = totalError/i;
70 successPercent = success/i;
71
72 consistency = sqrt(sum((error-averageError).^2));
73
74 saveName = [savePath '/' saveName];
75 % save(saveName, 'epsilon', 'averageError', 'successPercent', 'seed');
76
77 % figName = [savePath '/' 'GBestAccuracyFig'];
78 % saveas(1, [figName '.fig']);
79 % saveas(1, [figName '.jpg']);
80 % saveas(1, [figName '.pdf']);
81
82 index = randi(100,1);
83
84 %%%%%%% Figure 2 %%%%%% Plots a pBest progression. %%%%%%%
85 % pBest = cell2mat(pCell(index));
86 % figure(2); clf(2); hold on;
87 %
88 % Plot a circle.
89 % r = sqrt(epsilon);
90 % ang=0:0.01:2*pi;
91 % xp= r*cos(ang);
92 % yp= r*sin(ang);
93 % plot(0+xp,0+yp,'red');
```

```
%
% title(['Final pBest location for run' num2str(index)...
%        '. noise = ' num2str(noise)]);
% for i=1:length(pBest)
%     plot(pBest(i,1),pBest(i,2), 'black*');
% end
%
% figName = [savePath '/' 'PbestAccuracyFig'];
% saveas(2, [figName '.fig']);
% saveas(2, [figName '.jpg']);
% saveas(2, [figName '.pdf']);



% figure(3); clf(3); hold on;
% %title('gBest progression for all 100 MC runs');
% xlabel('Iteration Number');
% ylabel('Function Value');
%
% for i=1:length(gCell)
%     gBest = cell2mat(gCell(i));
%     gProg = zeros(length(gBest),1);
%     for j=1:length(gBest)
%         gProg(j) = Rastrigin(gBest(j,:));
%     end
%     plot(gProg);
% end
%
% figName = [savePath '/' 'GbestProgressionFig'];
% saveas(3, [figName '.fig']);
% saveas(3, [figName '.jpg']);
% saveas(3, [figName '.pdf']);


length = size(MCprog,2);
avg = zeros(1,length);
stddev = zeros(1, length);
for i=1:length
    avg(i) = mean(MCprog(:,i));
    stddev(i) = std(MCprog(:,i));
end

pTile = prctile(MCprog,[10,90],1);
lower = pTile(1,:);
upper = pTile(2,:);
maxIterations = 100;
X = 0:maxIterations-1;


% figure(4); clf(4); hold on;
% title('Distribution of the Fitness Value of the Global Best (1000 ...
    runs).');
% xlabel('Iteration Number');
% ylabel('Average Distribution');
```

```
147 % axis([min(X),max(X),-inf,inf]);  % Set axis.
148 %
149 % set(gca,'fontsize',18);  % Change the font size.
150 % title('Fitness of PSO');
151 %
152 % errorbar(avg,stddev);
153 % errorbar(X,avg,lower,upper);
154
155 % figName = [savePath '/' 'ErrorBarFig'];
156 % saveas(4, [figName '.fig']);
157 % saveas(4, [figName '.jpg']);
158 % saveas(4, [figName '.pdf']);
159
160 %
161 % figure(5); clf(5); hold on;
162 % hist(histData, 20);
163 % hist(histData, [0:0.05:1.2]);
164 % hist(histData, [0:0.07:2]);
165 % xlabel('Distance from global optimum.');
166 % ylabel('Number in bin. (/100)');
167 %
168 % axis([0,1.2,0,20]);
169 % xlim([0,1.2]);
170 % xlim([0,2]);
171 % set(gca,'fontsize',18);  % Change the font size.
172 % title('PSO Distance from Optimum');
173 %
174 % figName = [savePath '/' 'HistDistFig'];
175 % saveas(5, [figName '.fig']);
176 % saveas(5, [figName '.jpg']);
177 % saveas(5, [figName '.pdf']);
178 %
179 % figure(6); clf(6); hold on;
180 % hist(MCprog(:,end), 20);
181 % xlabel('Fitness value.');
182 % ylabel('Number in bin. (/100)');
183 %
184 % figName = [savePath '/' 'HistFitFig'];
185 % saveas(6, [figName '.fig']);
186 % saveas(6, [figName '.jpg']);
187 % saveas(6, [figName '.pdf']);
188
189 figure(7); clf(7); hold on;
190 xlabel('Iteration Number','FontSize', 20);
191 ylabel('Average Distribution','FontSize', 20);
192 title('Fitness of PSO','FontSize', 20);
193 axis([min(X),max(X),-inf,inf]);
194 errorbar(X,avg,lower,upper,'LineWidth',2);
195 axes('Position',[.5,.5,.4,.4]);
196 xbin = linspace(0,150,20);
197 % xbin = linspace(0,300,20);
198 hist(histData, xbin);
199 axis([0,150,-inf,inf]);
200 % axis([0,300,-inf,inf]);
```

```
xlabel('Distance from global optimum.', 'FontSize', 14);
ylabel('Number in bin. (/100)', 'FontSize', 14);

figName = [savePath '/' 'ErrorHistFig'];
saveas(7, [figName '.fig']);
saveas(7, [figName '.jpg']);
saveas(7, [figName '.pdf']);
```

SphericalErrorProcNPSO.m

```
clear; clc; close all; format long;

% Processes noisy data from the NPSO that used the Spherical function.

% Potential load paths for different noise levels.
% load('NPSOData/Spherical/Noise00/NoisyData.mat');
% load('NPSOData/Spherical/Noise10/NoisyData.mat');
% load('NPSOData/Spherical/Noise25/NoisyData.mat');
% load('NPSOData/Spherical/Noise50/NoisyData.mat');
% load('ScaledNPSOData/Spherical/Noise00/NoisyData.mat');
% load('ScaledNPSOData/Spherical/Noise10/NoisyData.mat');
% load('ScaledNPSOData/Spherical/Noise25/NoisyData.mat');
% load('ScaledNPSOData/Spherical/Noise50/NoisyData.mat');
% load('ScaledLDINPSOData/Spherical/Noise00/NoisyData.mat');
load('ScaledLDINPSOData/Spherical/Noise25/NoisyData.mat');
saveName = 'ProcessedData';

% benchmark = @(x) Spherical(x);
benchmark = @(x) SphericalNoiseScaled(x,0,scaleX,scaleY);

actual = 0;
epsilon = .01;
success = 0;
totalError = 0;

%This is for the final run only. Figure _ shows the final gbest of each MC.
% figure(1); clf(1);
% hold on;
% plot(p(:,1),p(:,2),'blue*');
% plot(n(:,1),n(:,2),'red.');
% title(['Final gBest Value']);
%
% figName = [savePath '/' 'NBestAccuracyFig'];
% saveas(1, [figName '.fig']);
% saveas(1, [figName '.jpg']);
% saveas(1, [figName '.pdf']);

% figure(2); clf(2);
% hold on;
k = 1;
for i=1:length(gCell)
    gBest = cell2mat(gCell(i));
    gProg = zeros(length(gBest),1);
```

```
    for j=1:length(gBest)
        gProg(j) = benchmark(gBest(j,:));
    end
%    plot(gProg);
    histRawData(k,:) = gBest(end,:);
    k = k + 1;
end
%
% xlabel('Iteration Number');
% ylabel('Function Value');
% figName = [savePath '/' 'GbestProgressionFig'];
% saveas(2, [figName '.fig']);
% saveas(2, [figName '.jpg']);
% saveas(2, [figName '.pdf']);
%
% figure(3); clf(3);
% hold on;
length = size(MCprog,2);
avg = zeros(1,length);
for i=1:length
    avg(i) = mean(MCprog(:,i));
end
pTile = prctile(MCprog,[10,90],1);
lower = pTile(1,:);
upper = pTile(2,:);
maxIterations = 100;
X = [1:maxIterations];
% errorbar(X,avg,lower,upper);
% xlabel('Iteration Number');
% ylabel('Average Distribution');
% axis([min(X),max(X),-inf,inf]);
%
% set(gca,'fontsize',18);  % Change the font size.
% title('Fitness of NPSO');
%
% figName = [savePath '/' 'ErrorBarFig'];
% saveas(3, [figName '.fig']);
% saveas(3, [figName '.jpg']);
% saveas(3, [figName '.pdf']);


% figure(4); clf(4); hold on;
%
% spaceLow = -5.12;
% spaceHigh = -spaceLow;
%
% [x,y] = meshgrid(spaceLow:.1:spaceHigh,spaceLow:.1:spaceHigh);
% [m,n] = size(x);
% z = zeros(m,n);
% for i=1:m
%     for j=1:n
%         z(i,j) = Spherical([x(i,j),y(i,j)]);
%     end
% end
```

```
%
% contour(x,y,z);
%
%
[m,n] = size(histRawData);
error = zeros(1,m);
for i=1:m
    error(i) = abs(benchmark(histRawData(i,:)) - actual);
    if  error < epsilon
%        plot(histRawData(i,1),histRawData(i,2),'red*');
        success = success + 1;
    else
%        plot(histRawData(i,1),histRawData(i,2),'black*');
    end
    histData(i) = sqrt(MCbest(i,1)^2 + MCbest(i,2)^2);
end

maxError = max(error);
minError = min(error);
totalError = sum(error);
averageError = totalError/i;

consistency = sqrt(sum((error-averageError).^2));

% figure(5); clf(5); hold on;
% axis([0,5,0,70]);
% xlim([0, 2.5e-11])
% hist(histData,20); %  47);  %
% xlabel('Distance from global optimum.');
% ylabel('Number in bin. (/100)');
%
% set(gca,'fontsize',18);  % Change the font size.
% title('NPSO Distance from Optimum');
%
% figName = [savePath '/' 'HistDistFig'];
% saveas(5, [figName '.fig']);
% saveas(5, [figName '.jpg']);
% saveas(5, [figName '.pdf']);

% figure(6); clf(6); hold on;
% hist(MCprog(:,end), 20);
% xlabel('Fitness value.');
% ylabel('Number in bin. (/100)');
%
% figName = [savePath '/' 'HistFitFig'];
% saveas(6, [figName '.fig']);
% saveas(6, [figName '.jpg']);
% saveas(6, [figName '.pdf']);

figure(7); clf(7); hold on;
xlabel('Iteration Number','FontSize', 20);
ylabel('Average Distribution','FontSize', 20);
title('Fitness of NPSO','FontSize', 20);
axis([min(X),max(X),-inf,inf]);
```

```
errorbar(X,avg,lower,upper,'LineWidth',2);
axes('Position',[.5,.5,.4,.4]);
xbin = linspace(0,2.5*10^(-9),20);
% xbin = linspace(0,25,20);
hist(histData, xbin);
axis([0,2.5*10^(-9),-inf,inf]);
% axis([0,25,-inf,inf]);
xlabel('Distance from global optimum.', 'FontSize', 14);
ylabel('Number in bin. (/100)', 'FontSize', 14);

figName = [savePath '/' 'ErrorHistFig'];
saveas(7, [figName '.fig']);
saveas(7, [figName '.jpg']);
saveas(7, [figName '.pdf']);
```

RastriginErrorProcNPSO.m

```
clear; clc; close all; format long;

% Processes noisy data from the NPSO that used the Rastrigin function.

% Potential load paths for different noise levels.
% load('NPSOData/Rastrigin/Noise00/NoisyData.mat');
% load('NPSOData/Rastrigin/Noise50/NoisyData.mat');
% load('NPSOData/Rastrigin/Noise100/NoisyData.mat');
% load('NPSOData/Rastrigin/Noise100_I2/NoisyData.mat');
% load('NPSOData/Rastrigin/Noise150/NoisyData.mat');
load('ScaledNPSOData/Rastrigin/Noise00/NoisyData.mat');
% load('ScaledNPSOData/Rastrigin/Noise50/NoisyData.mat');
% load('ScaledNPSOData/Rastrigin/Noise100/NoisyData.mat');
% load('ScaledNPSOData/Rastrigin/Noise100_I2/NoisyData.mat');
% load('ScaledNPSOData/Rastrigin/Noise150/NoisyData.mat');
% load('ScaledNPSOData/Rastrigin/Noise200/NoisyData.mat');
% load('ScaledNPSOData/Rastrigin/Noise400/NoisyData.mat');

% benchmark = @(x) Rastrigin(x);
benchmark = @(x) RastriginNoiseScaled(x,0,scaleX,scaleY,.25);

saveName = 'ProcessedData';

actual = 0;
epsilon = .01;
success = 0;
totalError = 0;

%This is for the final run only. Figure _ shows the final gbest of each MC.
% figure(1); clf(1);
% hold on;
% plot(p(:,1),p(:,2),'blue*');
% plot(n(:,1),n(:,2),'red.');
% title(['Final gBest Value']);
%
% figName = [savePath '/' 'NBestAccuracyFig'];
```

```
% saveas(1, [figName '.fig']);
% saveas(1, [figName '.jpg']);
% saveas(1, [figName '.pdf']);

% figure(2); clf(2);
% hold on;
k = 1;
for i=1:length(gCell)
    gBest = cell2mat(gCell(i));
    gProg = zeros(length(gBest),1);
    for j=1:length(gBest)
        gProg(j) = benchmark(gBest(j,:));
    end
%    plot(gProg);
    histRawData(k,:) = gBest(end,:);
    k = k + 1;
end
%
% xlabel('Iteration Number');
% ylabel('Function Value');
% figName = [savePath '/' 'GbestProgressionFig'];
% saveas(2, [figName '.fig']);
% saveas(2, [figName '.jpg']);
% saveas(2, [figName '.pdf']);

% figure(3); clf(3);
% hold on;
length = size(MCprog,2);
avg = zeros(1,length);
for i=1:length
    avg(i) = mean(MCprog(:,i));
end
pTile = prctile(MCprog,[10,90],1);
lower = pTile(1,:);
upper = pTile(2,:);
maxIterations = 100;
X = [1:maxIterations];
% errorbar(X,avg,lower,upper);
% xlabel('Iteration Number');
% ylabel('Average Distribution');
% axis([min(X),max(X),-inf,inf]);
%
% set(gca,'fontsize',18);  % Change the font size.
% title('Fitness of NPSO');
%
% figName = [savePath '/' 'ErrorBarFig'];
% saveas(3, [figName '.fig']);
% saveas(3, [figName '.jpg']);
% saveas(3, [figName '.pdf']);


% figure(4); clf(4); hold on;
%
% spaceLow = -5.12;
```

```
% spaceHigh = -spaceLow;
%
% [x,y] = meshgrid(spaceLow:.1:spaceHigh,spaceLow:.1:spaceHigh);
% [m,n] = size(x);
% z = zeros(m,n);
% for i=1:m
%     for j=1:n
%         z(i,j) = Rastrigin([x(i,j),y(i,j)]);
%     end
% end
%
% contour(x,y,z);
%
%
[m,n] = size(histRawData);
error = zeros(1,m);
for i=1:m
    error(i) = abs(benchmark(histRawData(i,:)) - actual);
    if  error < epsilon
%        plot(histRawData(i,1),histRawData(i,2),'red*');
        success = success + 1;
    else
%        plot(histRawData(i,1),histRawData(i,2),'black*');
    end
    histData(i) = sqrt(MCbest(i,1)^2 + MCbest(i,2)^2);
end

maxError = max(error);
minError = min(error);
totalError = sum(error);
averageError = totalError/i;

consistency = sqrt(sum((error-averageError).^2));
%
% figure(5); clf(5); hold on;
% axis([0,5,0,70]);
% xlim([0, 2.5e-11])
% hist(histData,20); %  47);  %
% xlabel('Distance from global optimum.');
% ylabel('Number in bin. (/100)');
%
% set(gca,'fontsize',18);  % Change the font size.
% title('NPSO Distance from Optimum');
%
% figName = [savePath '/' 'HistDistFig'];
% saveas(5, [figName '.fig']);
% saveas(5, [figName '.jpg']);
% saveas(5, [figName '.pdf']);

% figure(6); clf(6); hold on;
% hist(MCprog(:,end), 20);
% xlabel('Fitness value.');
% ylabel('Number in bin. (/100)');
%
```

```
% figName = [savePath '/' 'HistFitFig'];
% saveas(6, [figName '.fig']);
% saveas(6, [figName '.jpg']);
% saveas(6, [figName '.pdf']);

figure(7); clf(7); hold on;
xlabel('Iteration Number','FontSize', 20);
ylabel('Average Distribution','FontSize', 20);
title('Fitness of PSO','FontSize', 20);
axis([min(X),max(X),-inf,inf]);
errorbar(X,avg,lower,upper,'LineWidth',2);
axes('Position',[.5,.5,.4,.4]);
xbin = linspace(0,5*10^(-6),20);
% xbin = linspace(0,80,20);
% xbin = linspace(0,500,20);
hist(histData, xbin);
axis([0,5*10^(-6),-inf,inf]);
% axis([0,80,-inf,inf]);
% axis([0,500,-inf,inf]);
xlabel('Distance from global optimum.','FontSize', 14);
ylabel('Number in bin. (/100)','FontSize', 14);

figName = [savePath '/' 'ErrorHistFig'];
saveas(7, [figName '.fig']);
saveas(7, [figName '.jpg']);
saveas(7, [figName '.pdf']);
```

RosenbrockErrorProcNPSO.m

```
clear; clc; close all; format long;

% Processes noisy data from the NPSO that used the Rosenbrock function.

% Potential load paths for different noise levels.
% load('NPSOData/Rosenbrock/Noise00/NoisyData.mat');
% load('NPSOData/Rosenbrock/Noise10/NoisyData.mat');
% load('NPSOData/Rosenbrock/Noise25/NoisyData.mat');
% load('NPSOData/Rosenbrock/Noise50/NoisyData.mat');
load('ScaledNPSOData/Rosenbrock/Noise00/NoisyData.mat');
% load('ScaledNPSOData/Rosenbrock/Noise10/NoisyData.mat');
% load('ScaledNPSOData/Rosenbrock/Noise25/NoisyData.mat');
% load('ScaledNPSOData/Rosenbrock/Noise50/NoisyData.mat');
saveName = 'ProcessedData';

% benchmark = @(x) Rosenbrock(x);
benchmark = @(x) RosenbrockNoiseScaled(x,0,scaleX,scaleY);

actual = 0;
epsilon = .01;
success = 0;
totalError = 0;

%This is for the final run only. Figure _ shows the final gbest of each MC.
```

```
% figure(1); clf(1);
% hold on;
% plot(p(:,1),p(:,2),'blue*');
% plot(n(:,1),n(:,2),'red.');
% title(['Final gBest Value']);
%
% figName = [savePath '/' 'NBestAccuracyFig'];
% saveas(1, [figName '.fig']);
% saveas(1, [figName '.jpg']);
% saveas(1, [figName '.pdf']);

% figure(2); clf(2);
% hold on;
k = 1;
for i=1:length(gCell)
    gBest = cell2mat(gCell(i));
    gProg = zeros(length(gBest),1);
    for j=1:length(gBest)
        gProg(j) = benchmark(gBest(j,:));
    end
%    plot(gProg);
    histRawData(k,:) = gBest(end,:);
    k = k + 1;
end
%
% xlabel('Iteration Number');
% ylabel('Function Value');
% figName = [savePath '/' 'GbestProgressionFig'];
% saveas(2, [figName '.fig']);
% saveas(2, [figName '.jpg']);
% saveas(2, [figName '.pdf']);

% figure(3); clf(3);
% hold on;
length = size(MCprog,2);
avg = zeros(1,length);
for i=1:length
    avg(i) = mean(MCprog(:,i));
end
pTile = prctile(MCprog,[10,90],1);
lower = pTile(1,:);
upper = pTile(2,:);
maxIterations = 100;
X = [1:maxIterations];
% errorbar(X,avg,lower,upper);
% xlabel('Iteration Number');
% ylabel('Average Distribution');
% axis([min(X),max(X),-inf,inf]);
%
% set(gca,'fontsize',18);  % Change the font size.
% title('Fitness of NPSO');
%
% figName = [savePath '/' 'ErrorBarFig'];
% saveas(3, [figName '.fig']);
```

```
% saveas(3, [figName '.jpg']);
% saveas(3, [figName '.pdf']);


% figure(4); clf(4); hold on;
%
% spaceLow = -5.12;
% spaceHigh = -spaceLow;
%
% [x,y] = meshgrid(spaceLow:.1:spaceHigh,spaceLow:.1:spaceHigh);
% [m,n] = size(x);
% z = zeros(m,n);
% for i=1:m
%     for j=1:n
%         z(i,j) = Rosenbrock([x(i,j),y(i,j)]);
%     end
% end
%
% contour(x,y,z);
%
%
[m,n] = size(histRawData);
error = zeros(1,m);
for i=1:m
    error(i) = abs(benchmark(histRawData(i,:)) - actual);
    if  error < epsilon
%        plot(histRawData(i,1),histRawData(i,2),'red*');
        success = success + 1;
    else
%        plot(histRawData(i,1),histRawData(i,2),'black*');
    end
    histData(i) = sqrt((1*scaleX - MCbest(i,1))^2 + (1*scaleY - ...
       MCbest(i,2))^2);
end

maxError = max(error);
minError = min(error);
totalError = sum(error);
averageError = totalError/i;

consistency = sqrt(sum((error-averageError).^2));

% figure(5); clf(5); hold on;
% axis([0,5,0,70]);
% xlim([0, 2.5e-11])
% hist(histData,20); %  47);  %
% xlabel('Distance from global optimum.');
% ylabel('Number in bin. (/100)');
%
% set(gca,'fontsize',18);  % Change the font size.
% title('NPSO Distance from Optimum');
%
% figName = [savePath '/' 'HistDistFig'];
% saveas(5, [figName '.fig']);
```

```
% saveas(5, [figName '.jpg']);
% saveas(5, [figName '.pdf']);

% figure(6); clf(6); hold on;
% hist(MCprog(:,end), 20);
% xlabel('Fitness value.');
% ylabel('Number in bin. (/100)');
%
% figName = [savePath '/' 'HistFitFig'];
% saveas(6, [figName '.fig']);
% saveas(6, [figName '.jpg']);
% saveas(6, [figName '.pdf']);

figure(7); clf(7); hold on;
xlabel('Iteration Number','FontSize', 20);
ylabel('Average Distribution','FontSize', 20);
title('Fitness of NPSO','FontSize', 20);
axis([min(X),max(X),-inf,inf]);
errorbar(X,avg,lower,upper,'LineWidth',2);
axes('Position',[.5,.5,.4,.4]);
xbin = linspace(0,150,20);
% xbin = linspace(0,300,20);
hist(histData, xbin);
axis([0,150,-inf,inf]);
% axis([0,300,-inf,inf]);
xlabel('Distance from global optimum.', 'FontSize', 14);
ylabel('Number in bin. (/100)', 'FontSize', 14);

figName = [savePath '/' 'ErrorHistFig'];
saveas(7, [figName '.fig']);
saveas(7, [figName '.jpg']);
saveas(7, [figName '.pdf']);
```

APPENDIX D: Kilombo Source Code

D.1 Pseudo-Vector Motion (PVM)

lightFinder.h

```
1
2  #ifndef M_PI
3  #define M_PI 3.14159265358979323846264338327950288419716939937510582097494
4  #endif
5
6
7  // declare motion variable type
8  typedef enum {
9      STOP,
10     FORWARD,
11     LEFT,
12     RIGHT
13 } motion_t;
14
15 // declare state variable type
16 typedef enum {
17     BETTER,
18     WORSE,
19     NONE,
20     SUCCESS,
21 } search_state_t;
22
23 // declare variables
24
25 typedef struct
26 {
27   search_state_t search_state;
28   uint8_t cur_distance;
29   uint8_t new_message;
30   distance_measurement_t dist;
31
32   uint32_t intensity;
33 //  uint8_t nBest;
34 //  uint8_t nBestID;
35   uint8_t msgMeasurement;
36   uint8_t msgID;
37
38   uint32_t lastIntensity;
39   uint32_t lastTicks;
40   uint32_t stepTicks;
41
42   message_t transmit_msg;
```

```
} USERDATA;

extern USERDATA *mydata;
```

lightFinder.c

```
/*
 * Lightly modified to work in the simulator, in particular:
 * - mydata->variable for global variables
 * - callback function cb_botinfo() to report bot state back to the ...
    simulator for display
 * - spin-up motors only when required, using the helper function  ...
    smooth_set_motors()
 *
 */

#include <math.h>

#include <kilombo.h>

#include "lightFinder.h"

#ifdef SIMULATOR
#include <stdio.h> // for printf
#else
#include <avr/io.h>  // for microcontroller register defs
//  #define DEBUG          // for printf to serial port
//  #include "debug.h"
#endif

REGISTER_USERDATA(USERDATA)

// declare constants
static const uint8_t STEP_TIME = 10;
static const uint32_t TICKS_TO_SUCCESS = 300; //
static const uint8_t TOOCLOSE_DISTANCE = 40; //
//static const uint8_t TOOFAR_DISTANCE   = 65; //

/* Helper function for setting motor speed smoothly
 */
void smooth_set_motors(uint8_t ccw, uint8_t cw)
{
  // OCR2A = ccw;  OCR2B = cw;
#ifdef KILOBOT
  uint8_t l = 0, r = 0;
  if (ccw && !OCR2A) // we want left motor on, and it's off
    l = 0xff;
  if (cw && !OCR2B)  // we want right motor on, and it's off
    r = 0xff;
  if (l || r)        // at least one motor needs spin-up
    {
      set_motors(l, r);
      delay(15);
```

```
46      }
47  #endif
48    // spin-up is done, now we set the real value
49    set_motors(ccw, cw);
50    //delay(30);    PERHAPS WE CAN VARY THE AMOUNT WITH A DELAY
51  }
52
53
54  void set_motion(motion_t new_motion)
55  {
56    switch(new_motion) {
57    case STOP:
58      smooth_set_motors(0,0);
59      break;
60    case FORWARD:
61      smooth_set_motors(kilo_straight_left, kilo_straight_right);
62      break;
63    case LEFT:
64      smooth_set_motors(kilo_turn_left, 0);
65      break;
66    case RIGHT:
67      smooth_set_motors(0, kilo_turn_right);
68      break;
69    }
70  }
71
72  void stallCollision() {
73      if (kilo_uid < mydata->msgID) {
74          set_motion(FORWARD);
75      } else {
76          set_motion(STOP);
77      }
78  }
79
80  void loop() {
81  mydata->intensity = get_ambientlight();
82  if (kilo_ticks ≥ mydata->stepTicks + STEP_TIME) {
83  //    mydata->intensity = get_ambientlight();
84      if (mydata->intensity < mydata->lastIntensity) {
85      mydata->lastIntensity = mydata->intensity;
86          mydata->lastTicks = kilo_ticks;
87          mydata->search_state = BETTER;
88      } else if (mydata->intensity > mydata->lastIntensity) {
89      mydata->lastIntensity = mydata->intensity;
90          mydata->lastTicks = kilo_ticks;
91      mydata->search_state = WORSE;
92      } else {
93          if (kilo_ticks ≥ mydata->lastTicks + TICKS_TO_SUCCESS) {
94          mydata->search_state = SUCCESS;
95      } else {
96          mydata->search_state = NONE;
97          }
98      }
99      mydata->stepTicks = kilo_ticks;
```

```
100     //BASIC Light Finder
101     switch(mydata->search_state) {
102         case SUCCESS:
103         set_motion(STOP);
104             break;
105     case NONE:
106             set_motion(LEFT);
107         break;
108         case BETTER:
109             set_motion(FORWARD);
110         break;
111     case WORSE:
112             set_motion(LEFT);
113         break;
114    }
115 }
116
117
118     // VARIABLE BEING TRANSMITTED, CURRENTLY LIGHT INTENSITY.
119     mydata->transmit_msg.data[0] = mydata->intensity;//nBest;
120
121     // Update distance estimate with every message
122    if (mydata->new_message) {
123         mydata->new_message = 0;
124         mydata->cur_distance = estimate_distance(&mydata->dist);
125     }// else if (mydata->cur_distance == 0) // skip state machine if no ...
        distance measurement available
126 //        return;
127      if (mydata->cur_distance ≤ TOOCLOSE_DISTANCE) {
128 //  set_motion(STOP);
129         stallCollision();
130         return;
131     }
132 }
133
134
135 void message_rx(message_t *m, distance_measurement_t *d) {
136     mydata->new_message = 1;
137 //    mydata->new_message = m->data[0];
138     mydata->dist = *d;
139     mydata->msgMeasurement = m->data[0];
140     mydata->msgID = m->data[1];
141 }
142
143 void setup_message(void)
144 {
145   mydata->transmit_msg.type = NORMAL;
146 //  mydata->transmit_msg.data[0] = kilo_uid & 0xff; //low byte of ID, ...
      currently not really used for anything
147   mydata->transmit_msg.data[0] = mydata->intensity;
148   mydata->transmit_msg.data[1] = kilo_uid;
149   //finally, calculate a message check sum
150   mydata->transmit_msg.crc = message_crc(&mydata->transmit_msg);
151 }
```

```
152
153 message_t *message_tx()
154 {
155   return &mydata->transmit_msg;
156 }
157
158 void setup()
159 {
160   mydata->search_state = NONE;
161   mydata->cur_distance = 100;
162   mydata->new_message = 0;
163   mydata->intensity = 1023;
164
165 //  mydata->nBest = 255;//0;
166 //  mydata->nBestID = -1;
167 //  mydata->msgMeasurement = -1;
168
169   mydata->lastIntensity = 0;
170   mydata->lastTicks = 0;
171   mydata->stepTicks = 0;
172
173   setup_message();
174
175   set_color(RGB(0,3,0)); // color of the stationary bot
176 }
177
178
179 #ifdef SIMULATOR
180 /* provide a text string for the simulator status bar about this bot */
181 static char botinfo_buffer[10000];
182 char *cb_botinfo(void)
183 {
184   char *p = botinfo_buffer;
185   p += sprintf (p, "ID: %d \n", kilo_uid);
186 //  p += sprintf (p, "xCoord: %d",);
187 //  p += sprintf (p, "");
188   p += sprintf (p, "Light Intensity: %d \n", mydata->intensity);
189   p += sprintf (p, "Distance: %d \n", mydata->cur_distance);
190 //  p += sprintf (p, "nBest Intensity: %d \n", mydata->nBest);
191 //  p += sprintf (p, "nBest ID : %d \n", mydata->nBestID);
192 //  p += sprintf (p, "rxMsg Intensity: %d \n", mydata->msgMeasurement);
193 //  if (mydata->orbit_state == ORBIT_NORMAL)
194 //    p += sprintf (p, "State: ORBIT_NORMAL\n");
195 //  if (mydata->orbit_state == ORBIT_TOOCLOSE)
196 //    p += sprintf (p, "State: ORBIT_TOOCLOSE\n");
197
198   return botinfo_buffer;
199 }
200 //#endif
201
202 // ATTEMPTED LIGHTING CALLBACK
203 //int16_t cb_lighting(double xCoord, double yCoord)
204 //{
205 //    int16_t intensity = 1023 - sqrt(xCoord*xCoord + yCoord*yCoord);
```

```
206 //    int16_t intensity = get_ambientlight(); //BROKEN HERE
207 //    return intensity;
208 //}
209
210 #include <jansson.h>
211 json_t *json_state();
212 #endif
213
214 int main() {
215     kilo_init();
216     kilo_message_rx = message_rx;
217
218 //    SET_CALLBACK(botinfo, cb_botinfo);
219 //    SET_CALLBACK(lighting, cb_lighting);
220
221     // bot 0 is stationary and transmits messages. Other bots orbit ...
          around it.
222 //    if (kilo_uid == 0)
223     kilo_message_tx = message_tx;
224
225     kilo_start(setup, loop);
226
227     SET_CALLBACK(botinfo, cb_botinfo);
228     SET_CALLBACK(json_state, json_state);
229
230     return 0;
231 }
```

D.2 Cognitive PVM

smartLightFinder.h

```
1
2 #ifndef M_PI
3 #define M_PI 3.14159265358979323846264338327950288419716939937510582097494
4 #endif
5
6
7 // declare motion variable type
8 typedef enum {
9     STOP,
10    FORWARD,
11    LEFT,
12    RIGHT
13 } motion_t;
14
15 // declare state variable type
16 typedef enum {
17    //THESE STATES DESCRIBE INSTRUMENT MEASUREMENTS
18    BETTER,
19    WORSE,
```

```
20      NONE,
21      SUCCESS,
22      //THESE STATES DESCRIBE WAITING CONDITIONS
23  } search_state_t;
24
25  typedef enum {
26      WAIT,
27      TRY,
28      FAIL,
29      NO_WAIT,
30  } wait_state_t;
31
32  // declare variables
33
34  typedef struct
35  {
36    search_state_t search_state;
37    uint8_t cur_distance;
38    uint8_t new_message;
39    distance_measurement_t dist;
40    motion_t motionHistory[10];
41    search_state_t lastState;
42
43    uint16_t intensity;
44  //  uint8_t nBest;
45  //  uint8_t nBestID;
46    uint16_t msgMeasurement;
47    uint8_t msgID;
48
49    uint16_t lastIntensity;
50    uint32_t lastTicks;
51    uint32_t stepTicks;
52    uint16_t stepTime;
53    uint8_t moveCounter;
54
55    uint8_t waitID;
56    wait_state_t waitState;
57    wait_state_t msgWaitState;
58
59    message_t transmit_msg;
60  } USERDATA;
61
62  extern USERDATA *mydata;
```

smartLightFinder.c

```
1  /*
2   * Lightly modified to work in the simulator, in particular:
3   * - mydata->variable for global variables
4   * - callback function cb_botinfo() to report bot state back to the ...
        simulator for display
5   * - spin-up motors only when required, using the helper function  ...
        smooth_set_motors()
```

```
 *
 */

#include <math.h>

#include <kilombo.h>

#include "smartLightFinder.h"

#ifdef SIMULATOR
#include <stdio.h> // for printf
#else
#include <avr/io.h>  // for microcontroller register defs
//  #define DEBUG          // for printf to serial port
//  #include "debug.h"
#endif

REGISTER_USERDATA(USERDATA)

// declare constants
static const uint32_t TICKS_TO_SUCCESS = 300; //
static const uint8_t TOOCLOSE_DISTANCE = 40; //
static motion_t CURRENT_TURN = LEFT;
//static motion_t NEXT_MOTION = STOP;
static const uint16_t STEP_TIME = 10;
static const uint16_t QUARTER_TURN = 32;
static const uint16_t TURN_AROUND = 256;
//static const uint8_t TOOFAR_DISTANCE   = 65; //

/* Helper function for setting motor speed smoothly
 */
void smooth_set_motors(uint8_t ccw, uint8_t cw)
{
  // OCR2A = ccw;  OCR2B = cw;
#ifdef KILOBOT
  uint8_t l = 0, r = 0;
  if (ccw && !OCR2A) // we want left motor on, and it's off
    l = 0xff;
  if (cw && !OCR2B)  // we want right motor on, and it's off
    r = 0xff;
  if (l || r)        // at least one motor needs spin-up
    {
      set_motors(l, r);
      delay(15);
    }
#endif
  // spin-up is done, now we set the real value
  set_motors(ccw, cw);
  //delay(30);   PERHAPS WE CAN VARY THE AMOUNT WITH A DELAY
}


void set_motion(motion_t new_motion)
{
```

```
60    switch(new_motion) {
61    case STOP:
62      smooth_set_motors(0,0);
63      break;
64    case FORWARD:
65      smooth_set_motors(kilo_straight_left, kilo_straight_right);
66      break;
67    case LEFT:
68      smooth_set_motors(kilo_turn_left, 0);
69      break;
70    case RIGHT:
71      smooth_set_motors(0, kilo_turn_right);
72      break;
73    }
74  }
75
76  void updateMeasurements() {
77      mydata->intensity = get_ambientlight();
78      if (mydata->intensity < mydata->lastIntensity) {
79      mydata->lastIntensity = mydata->intensity;
80          mydata->lastTicks = kilo_ticks;
81          mydata->search_state = BETTER;
82      } else if (mydata->intensity > mydata->lastIntensity) {
83      mydata->lastIntensity = mydata->intensity;
84          mydata->lastTicks = kilo_ticks;
85      mydata->search_state = WORSE;
86      } else {
87          if (kilo_ticks ≥ mydata->lastTicks + TICKS_TO_SUCCESS) {
88          mydata->search_state = SUCCESS;
89      } else {
90          mydata->search_state = NONE;
91          }
92      }
93  }
94
95  void updateMotionHistory(motion_t new_motion) {
96      for (int i = 1; i < 10; i++) {
97          mydata->motionHistory[i] = mydata->motionHistory[i-1];
98      }
99      mydata->motionHistory[0] = new_motion;
100 }
101
102 void changeTurn() {
103     if (CURRENT_TURN == LEFT) {
104     CURRENT_TURN = RIGHT;
105     } else {
106     CURRENT_TURN = LEFT;
107     }
108 }
109
110 void setStepTime(uint16_t stepTime) {
111     mydata->stepTime = stepTime;
112 }
113
```

```
114 void resetStepTime() {
115     mydata->stepTime = STEP_TIME;
116 }
117
118 void checkNONEcondition() {
119     if(mydata->moveCounter == 3) {
120     set_motion(CURRENT_TURN);
121 //        NEXT_MOTION = CURRENT_TURN;
122         updateMotionHistory(CURRENT_TURN);
123     mydata->moveCounter = 0;
124     } else {
125         set_motion(FORWARD);
126 //  NEXT_MOTION = FORWARD;
127         updateMotionHistory(FORWARD);
128     mydata->moveCounter++;
129     }
130 }
131
132 void checkBETTERcondition() {
133     set_motion(FORWARD);
134 //    NEXT_MOTION = FORWARD;
135     updateMotionHistory(FORWARD);
136     mydata->moveCounter = 0;
137 //    return FORWARD;
138 }
139
140 void checkWORSEcondition() {
141     if (mydata->lastState != WORSE &&
142     mydata->motionHistory[0] == CURRENT_TURN) {
143     changeTurn();
144     }// else if (mydata->lastState == WORSE &&
145 //         mydata->motionHistory[0] == CURRENT_TURN) {
146 //  setStepTime(QUARTER_TURN);
147 //    } else {
148     setStepTime(TURN_AROUND);
149 //    }
150     set_motion(CURRENT_TURN);
151 //    NEXT_MOTION = CURRENT_TURN;
152     updateMotionHistory(CURRENT_TURN);
153 }
154
155 void stallCollision() {
156 //    if (mydata->waitID == 255) {
157         mydata->waitID = mydata->msgID;
158         mydata->lastTicks = kilo_ticks;
159 //    }
160     if (kilo_uid < mydata->waitID && mydata->waitState == NO_WAIT) {
161         set_motion(FORWARD);
162 //        NEXT_MOTION = FORWARD;
163 //    } else if (kilo_uid > mydata->waitID && mydata->msgWaitState == ...
    FAIL){
164 //  set_motion(FORWARD);
165 //    } else if (mydata->waitState == FAIL && mydata->msgWaitState == ...
    FAIL) {
```

```
166 //        setStepTime(TURN_AROUND);
167 //  set_motion(CURRENT_TURN);
168     } else {
169     set_motion(STOP);
170 //  NEXT_MOTION = STOP;
171     }
172 //        mydata->waitState = WAIT;
173 //}
174 
175 }
176 //    if (kilo_uid > mydata->msgID) {
177 //        resetStepTime();
178 //  set_motion(STOP);
179 //    } else {
180 //  resetStepTime();
181 //  set_motion(FORWARD);
182 //    }
183 
184 
185 void stateMachine() {
186     //SMARTER Light Finder (SELF COGNIZANT)
187     switch(mydata->search_state) {
188         case SUCCESS:
189         set_motion(STOP);
190 //      NEXT_MOTION = STOP;
191             updateMotionHistory(STOP);
192         break;
193     case NONE:
194         checkNONEcondition();
195         break;
196         case BETTER:
197         checkBETTERcondition();
198         break;
199     case WORSE:
200         checkWORSEcondition();
201             break;
202     }
203 }
204 
205 
206 void loop() {
207 if (mydata->search_state != SUCCESS) {
208     updateMeasurements();
209 }
210 //ONLY UPDATE MEASUREMENTS IF STEP TIME IS DONE.
211 if (kilo_ticks ≥ mydata->stepTicks + mydata->stepTime) {
212     resetStepTime();
213 //    if (mydata->search_state != SUCCESS) {
214 //        updateMeasurements();
215 //    }
216 //    mydata->lastTicks = kilo_ticks;
217     mydata->stepTicks = kilo_ticks;
218     stateMachine();
219     mydata->lastState = mydata->search_state;
```

```
220   }
221     // VARIABLE BEING TRANSMITTED, CURRENTLY LIGHT INTENSITY.
222     mydata->transmit_msg.data[0] = mydata->intensity;//nBest;
223     // Update distance estimate with every message
224     if (mydata->new_message) {
225         mydata->new_message = 0;
226         mydata->cur_distance = estimate_distance(&mydata->dist);
227     }
228     //TRY TO HANDLE COLLISION
229     if (mydata->cur_distance ≤ TOOCLOSE_DISTANCE && ...
            mydata->search_state != SUCCESS) {
230     stallCollision();
231     } else if (mydata->waitID == mydata->msgID) {
232     mydata->waitID = 255;
233     }
234 //set_motion(NEXT_MOTION);
235 }
236
237
238 void message_rx(message_t *m, distance_measurement_t *d) {
239     mydata->new_message = 1;
240 //    mydata->new_message = m->data[0];
241     mydata->dist = *d;
242     mydata->msgMeasurement = m->data[0];
243     mydata->msgID = m->data[1];
244     mydata->msgWaitState = m->data[2];
245 }
246
247 void setup_message(void)
248 {
249   mydata->transmit_msg.type = NORMAL;
250 //  mydata->transmit_msg.data[0] = kilo_uid & 0xff; //low byte of ID, ...
       currently not really used for anything
251   mydata->transmit_msg.data[0] = mydata->intensity;
252   mydata->transmit_msg.data[1] = kilo_uid;
253   mydata->transmit_msg.data[2] = mydata->waitState;
254   //finally, calculate a message check sum
255   mydata->transmit_msg.crc = message_crc(&mydata->transmit_msg);
256 }
257
258 message_t *message_tx()
259 {
260   return &mydata->transmit_msg;
261 }
262
263 void setup()
264 {
265   mydata->search_state = NONE;
266   mydata->cur_distance = 100;
267   mydata->new_message = 0;
268   mydata->intensity = 1023;
269
270 //  mydata->nBest = 255;//0;
271 //  mydata->nBestID = -1;
```

```
272 //  mydata->msgMeasurement = -1;
273
274   mydata->lastIntensity = 1023;
275   mydata->lastTicks = 0;
276   mydata->stepTicks = 0;
277   mydata->moveCounter = 0;
278
279   mydata->waitID = 255;
280   mydata->waitState = NO_WAIT;
281   mydata->msgWaitState = NO_WAIT;
282
283   setup_message();
284
285   set_color(RGB(0,3,0)); // color of the stationary bot
286 }
287
288
289 #ifdef SIMULATOR
290 /* provide a text string for the simulator status bar about this bot */
291 static char botinfo_buffer[10000];
292 char *cb_botinfo(void)
293 {
294   char *p = botinfo_buffer;
295   p += sprintf (p, "ID: %d \n", kilo_uid);
296 //  p += sprintf (p, "xCoord: %d",);
297 //  p += sprintf (p, "");
298   p += sprintf (p, "Light Intensity: %d \n", mydata->intensity);
299   p += sprintf (p, "Succces: %d \n", mydata->search_state);
300 //  p += sprintf (p, "Distance: %d from %d. \n", mydata->cur_distance, ...
      mydata->msgID);
301 //  p += sprintf (p, "Wait ID: %d", mydata->waitID);
302 //  p += sprintf (p, "nBest Intensity: %d \n", mydata->nBest);
303 //  p += sprintf (p, "nBest ID : %d \n", mydata->nBestID);
304 //  p += sprintf (p, "rxMsg Intensity: %d \n", mydata->msgMeasurement);
305 //  if (mydata->orbit_state == ORBIT_NORMAL)
306 //    p += sprintf (p, "State: ORBIT_NORMAL\n");
307 //  if (mydata->orbit_state == ORBIT_TOOCLOSE)
308 //    p += sprintf (p, "State: ORBIT_TOOCLOSE\n");
309
310   return botinfo_buffer;
311 }
312 //#endif
313
314 // ATTEMPTED LIGHTING CALLBACK
315 //int16_t cb_lighting(double xCoord, double yCoord)
316 //{
317 //    int16_t intensity = 1023 - sqrt(xCoord*xCoord + yCoord*yCoord);
318 //    int16_t intensity = get_ambientlight(); //BROKEN HERE
319 //    return intensity;
320 //}
321
322 #include <jansson.h>
323 json_t *json_state();
324 #endif
```

```
325
326
327 int main() {
328     kilo_init();
329     kilo_message_rx = message_rx;
330
331
332     // bot 0 is stationary and transmits messages. Other bots orbit ...
            around it.
333 //    if (kilo_uid == 0)
334     kilo_message_tx = message_tx;
335
336     kilo_start(setup, loop);
337
338     SET_CALLBACK(botinfo, cb_botinfo);
339 //    SET_CALLBACK(lighting, cb_lighting);
340     SET_CALLBACK(json_state, json_state);
341
342     return 0;
343 }
```

D.3 Social-CPVM

commLightFinder.h

```
1
2 #ifndef M_PI
3 #define M_PI 3.14159265358979323846264338327950288419716939937510582097494
4 #endif
5
6
7 // declare motion variable type
8 typedef enum {
9     STOP,
10    FORWARD,
11    LEFT,
12    RIGHT
13 } motion_t;
14
15 // declare state variable type
16 typedef enum {
17    //THESE STATES DESCRIBE INSTRUMENT MEASUREMENTS
18    BETTER,
19    WORSE,
20    NONE,
21    SUCCESS,
22    //THESE STATES DESCRIBE WAITING CONDITIONS
23 } search_state_t;
24
25 typedef enum {
26    WAIT,
27    TRY,
```

```
    FAIL,
    NO_WAIT,
} wait_state_t;

// declare variables

typedef struct
{
  search_state_t search_state;
  uint8_t cur_distance;
  uint8_t new_message;
  distance_measurement_t dist;
  motion_t motionHistory[10];
  search_state_t lastState;

  uint16_t intensity;
  uint16_t nBest;
  uint8_t nBestID;
  uint16_t msgMeasurement;
  uint8_t msgID;

  uint16_t lastIntensity;
  uint32_t lastTicks;
  uint32_t stepTicks;
  uint16_t stepTime;
  uint8_t moveCounter;

  uint8_t waitID;
  wait_state_t waitState;
  wait_state_t msgWaitState;

  search_state_t msgSuccess;
  uint16_t ticks_to_success;

  message_t transmit_msg;
} USERDATA;

extern USERDATA *mydata;
```

commLightFinder.c

```
/*
 * Lightly modified to work in the simulator, in particular:
 * - mydata->variable for global variables
 * - callback function cb_botinfo() to report bot state back to the ...
    simulator for display
 * - spin-up motors only when required, using the helper function  ...
    smooth_set_motors()
 *
 */

#include <math.h>

```

```
#include <kilombo.h>

#include "commLightFinder.h"

#ifdef SIMULATOR
#include <stdio.h> // for printf
#else
#include <avr/io.h>  // for microcontroller register defs
//  #define DEBUG          // for printf to serial port
//  #include "debug.h"
#endif

REGISTER_USERDATA(USERDATA)

// declare constants
static const uint32_t TICKS_TO_SUCCESS = 300; //
static const uint8_t TOOCLOSE_DISTANCE = 40; //
static motion_t CURRENT_TURN = LEFT;
//static motion_t NEXT_MOTION = STOP;
static const uint16_t STEP_TIME = 10;
static const uint16_t QUARTER_TURN = 32;
static const uint16_t TURN_AROUND = 256;
//static const uint8_t TOOFAR_DISTANCE   = 65; //

/* Helper function for setting motor speed smoothly
 */
void smooth_set_motors(uint8_t ccw, uint8_t cw)
{
  // OCR2A = ccw;  OCR2B = cw;
#ifdef KILOBOT
  uint8_t l = 0, r = 0;
  if (ccw && !OCR2A) // we want left motor on, and it's off
    l = 0xff;
  if (cw && !OCR2B)  // we want right motor on, and it's off
    r = 0xff;
  if (l || r)        // at least one motor needs spin-up
    {
      set_motors(l, r);
      delay(15);
    }
#endif
  // spin-up is done, now we set the real value
  set_motors(ccw, cw);
  //delay(30);   PERHAPS WE CAN VARY THE AMOUNT WITH A DELAY
}


void set_motion(motion_t new_motion)
{
  switch(new_motion) {
  case STOP:
    smooth_set_motors(0,0);
    break;
  case FORWARD:
```

```
65      smooth_set_motors(kilo_straight_left, kilo_straight_right);
66      break;
67    case LEFT:
68      smooth_set_motors(kilo_turn_left, 0);
69      break;
70    case RIGHT:
71      smooth_set_motors(0, kilo_turn_right);
72      break;
73    }
74  }
75
76  void updateMeasurements() {
77      mydata->intensity = get_ambientlight();
78      if (mydata->intensity < mydata->nBest) {
79          mydata->nBest = mydata->intensity;
80          mydata->nBestID = kilo_uid;
81      }
82      if (mydata->search_state != SUCCESS) {
83          if (mydata->intensity < mydata->lastIntensity) {
84              mydata->lastIntensity = mydata->intensity;
85              mydata->lastTicks = kilo_ticks;
86              mydata->search_state = BETTER;
87          } else if (mydata->intensity > mydata->lastIntensity) {
88          mydata->lastIntensity = mydata->intensity;
89              mydata->lastTicks = kilo_ticks;
90          mydata->search_state = WORSE;
91          } else {
92              if (kilo_ticks ≥ mydata->lastTicks + TICKS_TO_SUCCESS       ...
                    && mydata->intensity == mydata->nBest) {
93              mydata->search_state = SUCCESS;
94          } else {
95              mydata->search_state = NONE;
96              }
97      }
98      }
99  }
100
101 void updateMotionHistory(motion_t new_motion) {
102     for (int i = 1; i < 10; i++) {
103         mydata->motionHistory[i] = mydata->motionHistory[i-1];
104     }
105     mydata->motionHistory[0] = new_motion;
106 }
107
108 void changeTurn() {
109     if (CURRENT_TURN == LEFT) {
110     CURRENT_TURN = RIGHT;
111     } else {
112     CURRENT_TURN = LEFT;
113     }
114 }
115
116 void setStepTime(uint16_t stepTime) {
117     mydata->stepTime = stepTime;
```

```
118 }
119
120 void resetStepTime() {
121     mydata->stepTime = STEP_TIME;
122 }
123
124 void checkNONEcondition() {
125     if(mydata->moveCounter == 3) {
126     set_motion(CURRENT_TURN);
127 //         NEXT_MOTION = CURRENT_TURN;
128         updateMotionHistory(CURRENT_TURN);
129     mydata->moveCounter = 0;
130     } else {
131         set_motion(FORWARD);
132 //  NEXT_MOTION = FORWARD;
133         updateMotionHistory(FORWARD);
134     mydata->moveCounter++;
135     }
136 }
137
138 void checkBETTERcondition() {
139     set_motion(FORWARD);
140 //    NEXT_MOTION = FORWARD;
141     updateMotionHistory(FORWARD);
142     mydata->moveCounter = 0;
143 //    return FORWARD;
144 }
145
146 void checkWORSEcondition() {
147     if (mydata->lastState != WORSE &&
148     mydata->motionHistory[0] == CURRENT_TURN) {
149     changeTurn();
150     }// else if (mydata->lastState == WORSE &&
151 //         mydata->motionHistory[0] == CURRENT_TURN) {
152 //  setStepTime(QUARTER_TURN);
153 //    } else {
154     setStepTime(TURN_AROUND);
155 //    }
156     set_motion(CURRENT_TURN);
157 //    NEXT_MOTION = CURRENT_TURN;
158     updateMotionHistory(CURRENT_TURN);
159 }
160
161 void stallCollision() {
162 //    if (mydata->waitID == 255) {
163         mydata->waitID = mydata->msgID;
164         mydata->lastTicks = kilo_ticks;
165 //    }
166     if (kilo_uid < mydata->waitID && mydata->waitState == NO_WAIT) {
167         set_motion(FORWARD);
168 //        NEXT_MOTION = FORWARD;
169 //    } else if (kilo_uid > mydata->waitID && mydata->msgWaitState == ...
    FAIL){
170 //  set_motion(FORWARD);
```

```
171 //    } else if (mydata->waitState == FAIL && mydata->msgWaitState == ...
      FAIL) {
172 //        setStepTime(TURN_AROUND);
173 //  set_motion(CURRENT_TURN);
174     } else {
175     set_motion(STOP);
176 //  NEXT_MOTION = STOP;
177     }
178 //        mydata->waitState = WAIT;
179 //}
180
181 }
182 //    if (kilo_uid > mydata->msgID) {
183 //        resetStepTime();
184 //  set_motion(STOP);
185 //    } else {
186 //  resetStepTime();
187 //  set_motion(FORWARD);
188 //    }
189
190
191 void stateMachine() {
192     //SMARTER Light Finder (SELF COGNIZANT)
193     switch(mydata->search_state) {
194         case SUCCESS:
195         set_motion(STOP);
196 //      NEXT_MOTION = STOP;
197             updateMotionHistory(STOP);
198         break;
199     case NONE:
200         checkNONEcondition();
201         break;
202         case BETTER:
203         checkBETTERcondition();
204         break;
205     case WORSE:
206         checkWORSEcondition();
207             break;
208     }
209 }
210
211
212 void loop() {
213
214 //if (mydata->search_state != SUCCESS) {
215     updateMeasurements();
216 mydata->transmit_msg.data[0] = mydata->intensity;
217 mydata->transmit_msg.data[3] = mydata->search_state;
218 //}
219 //ONLY UPDATE MEASUREMENTS IF STEP TIME IS DONE.
220 if (kilo_ticks ≥ mydata->stepTicks + mydata->stepTime) {// &&
221 //    mydata->search_state != SUCCESS) {
222     resetStepTime();
223 //    if (mydata->search_state != SUCCESS) {
```

```
224 //         updateMeasurements();
225 //     }
226 //     mydata->lastTicks = kilo_ticks;
227     mydata->stepTicks = kilo_ticks;
228     stateMachine();
229     mydata->lastState = mydata->search_state;
230   }
231     // VARIABLE BEING TRANSMITTED, CURRENTLY LIGHT INTENSITY.
232 //     mydata->transmit_msg.data[0] = mydata->intensity;//nBest;
233     // Update distance estimate with every message
234     if (mydata->new_message) {
235         mydata->new_message = 0;
236         mydata->cur_distance = estimate_distance(&mydata->dist);
237         if (mydata->msgMeasurement < mydata->nBest) {
238             mydata->nBestID = mydata->msgID;
239         mydata->nBest = mydata->msgMeasurement;
240     }
241     }
242     //TRY TO HANDLE COLLISION
243     if (mydata->cur_distance ≤ TOOCLOSE_DISTANCE && ...
        mydata->search_state != SUCCESS && mydata->msgSuccess != ...
        SUCCESS) {
244     stallCollision();
245     } // IF COLLIDING WITH SUCCESS
246       else if (mydata->cur_distance ≤ TOOCLOSE_DISTANCE && ...
           mydata->search_state != SUCCESS && mydata->msgSuccess == ...
           SUCCESS) {
247         set_motion(STOP);
248         mydata->search_state = SUCCESS;
249         updateMotionHistory(STOP);
250     }
251 //set_motion(NEXT_MOTION);
252 }
253
254
255 void message_rx(message_t *m, distance_measurement_t *d) {
256     mydata->new_message = 1;
257 //     mydata->new_message = m->data[0];
258     mydata->dist = *d;
259     mydata->msgMeasurement = m->data[0];
260     mydata->msgID = m->data[1];
261     mydata->msgWaitState = m->data[2];
262     mydata->msgSuccess = m->data[3];
263 }
264
265 void setup_message(void)
266 {
267   mydata->transmit_msg.type = NORMAL;
268 //  mydata->transmit_msg.data[0] = kilo_uid & 0xff; //low byte of ID, ...
    currently not really used for anything
269   mydata->transmit_msg.data[0] = mydata->intensity;
270   mydata->transmit_msg.data[1] = kilo_uid;
271   mydata->transmit_msg.data[2] = mydata->waitState;
272   mydata->transmit_msg.data[3] = mydata->search_state;
```

```
273   //finally, calculate a message check sum
274   mydata->transmit_msg.crc = message_crc(&mydata->transmit_msg);
275 }
276
277 message_t *message_tx()
278 {
279   return &mydata->transmit_msg;
280 }
281
282 void setup()
283 {
284   mydata->search_state = NONE;
285   mydata->cur_distance = 100;
286   mydata->new_message = 0;
287   mydata->intensity = 1023;
288
289 //  mydata->nBest = 255;//0;
290 //  mydata->nBestID = -1;
291 //  mydata->msgMeasurement = -1;
292
293   mydata->lastIntensity = 1023;
294   mydata->lastTicks = 0;
295   mydata->stepTicks = 0;
296   mydata->moveCounter = 0;
297
298   mydata->waitID = 255;
299   mydata->waitState = NO_WAIT;
300   mydata->msgWaitState = NO_WAIT;
301   mydata->msgSuccess = NONE;
302
303   setup_message();
304
305   set_color(RGB(0,3,0)); // color of the stationary bot
306 }
307
308
309 #ifdef SIMULATOR
310 /* provide a text string for the simulator status bar about this bot */
311 static char botinfo_buffer[10000];
312 char *cb_botinfo(void)
313 {
314   char *p = botinfo_buffer;
315   p += sprintf (p, "ID: %d \n", kilo_uid);
316 //  p += sprintf (p, "xCoord: %d",);
317 //  p += sprintf (p, "");
318   p += sprintf (p, "Light Intensity: %d \n", mydata->intensity);
319   p += sprintf (p, "Succces: %d \n", mydata->search_state);
320 //  p += sprintf (p, "Distance: %d from %d. \n", mydata->cur_distance, ...
      mydata->msgID);
321 //  p += sprintf (p, "Wait ID: %d", mydata->waitID);
322 //  p += sprintf (p, "nBest Intensity: %d \n", mydata->nBest);
323 //  p += sprintf (p, "nBest ID : %d \n", mydata->nBestID);
324 //  p += sprintf (p, "rxMsg Intensity: %d \n", mydata->msgMeasurement);
325 //  if (mydata->orbit_state == ORBIT_NORMAL)
```

```
326 //    p += sprintf (p, "State: ORBIT_NORMAL\n");
327 //  if (mydata->orbit_state == ORBIT_TOOCLOSE)
328 //    p += sprintf (p, "State: ORBIT_TOOCLOSE\n");
329
330   return botinfo_buffer;
331 }
332 //#endif
333
334 // ATTEMPTED LIGHTING CALLBACK
335 //int16_t cb_lighting(double xCoord, double yCoord)
336 //{
337 //    int16_t intensity = 1020;
338 //    int16_t intensity = get_ambientlight(); //BROKEN HERE
339 //    printf("Intensity at %f, %f: %d\n", xCoord, yCoord, intensity);
340 //    return intensity;
341 //}
342
343 #include <jansson.h>
344 json_t *json_state();
345 #endif
346
347
348 int main() {
349     kilo_init();
350     kilo_message_rx = message_rx;
351
352
353     // bot 0 is stationary and transmits messages. Other bots orbit ...
            around it.
354 //    if (kilo_uid == 0)
355     kilo_message_tx = message_tx;
356
357     kilo_start(setup, loop);
358
359     SET_CALLBACK(botinfo, cb_botinfo);
360 //    SET_CALLBACK(lighting, cb_lighting);
361     SET_CALLBACK(json_state, json_state);
362
363     return 0;
364 }
```

D.4 Cummulative S-CPVM

bestLightFinder.h

```
1
2 #ifndef M_PI
3 #define M_PI 3.14159265358979323846264338327950288419716939937510582097494
4 #endif
5
6
7 // declare motion variable type
8 typedef enum {
```

```
    STOP,
    FORWARD,
    LEFT,
    RIGHT
} motion_t;

// declare state variable type
typedef enum {
    //THESE STATES DESCRIBE INSTRUMENT MEASUREMENTS
    BETTER,
    WORSE,
    NONE,
    SUCCESS,
    //THESE STATES DESCRIBE WAITING CONDITIONS
} search_state_t;

typedef enum {
    WAIT,
    TRY,
    FAIL,
    NO_WAIT,
} wait_state_t;

// declare variables

typedef struct
{
  search_state_t search_state;
  uint8_t cur_distance;
  uint8_t new_message;
  distance_measurement_t dist;
  motion_t motionHistory[10];
  search_state_t lastState;

  uint16_t intensity;
  uint16_t nBest;
  uint8_t nBestID;
  uint16_t msgMeasurement;
  uint8_t msgID;

  uint16_t lastIntensity;
  uint32_t lastTicks;
  uint32_t stepTicks;
  uint16_t stepTime;
  uint8_t moveCounter;

  uint8_t waitID;
  wait_state_t waitState;
  wait_state_t msgWaitState;

  search_state_t msgSuccess;
  uint16_t ticks_to_success;

  message_t transmit_msg;
```

```
} USERDATA;

extern USERDATA *mydata;
```

bestLightFinder.c

```
/*
 * Lightly modified to work in the simulator, in particular:
 * - mydata->variable for global variables
 * - callback function cb_botinfo() to report bot state back to the ...
    simulator for display
 * - spin-up motors only when required, using the helper function  ...
    smooth_set_motors()
 *
 */

#include <math.h>

#include <kilombo.h>

#include "bestLightFinder.h"

#ifdef SIMULATOR
#include <stdio.h> // for printf
#else
#include <avr/io.h>  // for microcontroller register defs
//  #define DEBUG          // for printf to serial port
//  #include "debug.h"
#endif

REGISTER_USERDATA(USERDATA)

// declare constants
static const uint16_t TICKS_TO_SUCCESS = 300; //
static const uint16_t REDUCTION_TIME = 3500;
static const uint8_t TOOCLOSE_DISTANCE = 40; //
static motion_t CURRENT_TURN = LEFT;
//static motion_t NEXT_MOTION = STOP;
static const uint16_t STEP_TIME = 10;
static const uint16_t QUARTER_TURN = 32;
static const uint16_t TURN_AROUND = 256;
//static const uint8_t TOOFAR_DISTANCE   = 65; //

/* Helper function for setting motor speed smoothly
 */
void smooth_set_motors(uint8_t ccw, uint8_t cw)
{
  // OCR2A = ccw;  OCR2B = cw;
#ifdef KILOBOT
  uint8_t l = 0, r = 0;
  if (ccw && !OCR2A) // we want left motor on, and it's off
    l = 0xff;
  if (cw && !OCR2B)  // we want right motor on, and it's off
```

```
46      r = 0xff;
47    if (l || r)         // at least one motor needs spin-up
48      {
49        set_motors(l, r);
50        delay(15);
51      }
52  #endif
53    // spin-up is done, now we set the real value
54    set_motors(ccw, cw);
55    //delay(30);   PERHAPS WE CAN VARY THE AMOUNT WITH A DELAY
56  }
57
58
59  void set_motion(motion_t new_motion)
60  {
61    switch(new_motion) {
62    case STOP:
63      smooth_set_motors(0,0);
64      break;
65    case FORWARD:
66      smooth_set_motors(kilo_straight_left, kilo_straight_right);
67      break;
68    case LEFT:
69      smooth_set_motors(kilo_turn_left, 0);
70      break;
71    case RIGHT:
72      smooth_set_motors(0, kilo_turn_right);
73      break;
74    }
75  }
76
77  void updateMeasurements() {
78      mydata->intensity = get_ambientlight();
79      if (mydata->intensity < mydata->nBest) {
80          mydata->nBest = mydata->intensity;
81          mydata->nBestID = kilo_uid;
82      }
83      if (mydata->search_state != SUCCESS) {
84          if (mydata->intensity < mydata->lastIntensity) {
85              mydata->lastIntensity = mydata->intensity;
86              mydata->lastTicks = kilo_ticks;
87              mydata->search_state = BETTER;
88          } else if (mydata->intensity > mydata->lastIntensity) {
89          mydata->lastIntensity = mydata->intensity;
90              mydata->lastTicks = kilo_ticks;
91          mydata->search_state = WORSE;
92          } else {
93              if (kilo_ticks ≥ mydata->lastTicks + ...
                    mydata->ticks_to_success && mydata->intensity == ...
                    mydata->nBest) {
94              mydata->search_state = SUCCESS;
95          } else {
96              mydata->search_state = NONE;
97              }
```

```
98      }
99      }
100 }
101
102 void updateMotionHistory(motion_t new_motion) {
103     for (int i = 1; i < 10; i++) {
104         mydata->motionHistory[i] = mydata->motionHistory[i-1];
105     }
106     mydata->motionHistory[0] = new_motion;
107 }
108
109 void changeTurn() {
110     if (CURRENT_TURN == LEFT) {
111     CURRENT_TURN = RIGHT;
112     } else {
113     CURRENT_TURN = LEFT;
114     }
115 }
116
117 void setStepTime(uint16_t stepTime) {
118     mydata->stepTime = stepTime;
119 }
120
121 void resetStepTime() {
122     mydata->stepTime = STEP_TIME;
123 }
124
125 void checkNONEcondition() {
126     if(mydata->moveCounter == 3) {
127     set_motion(CURRENT_TURN);
128 //         NEXT_MOTION = CURRENT_TURN;
129         updateMotionHistory(CURRENT_TURN);
130     mydata->moveCounter = 0;
131     } else {
132         set_motion(FORWARD);
133 //  NEXT_MOTION = FORWARD;
134         updateMotionHistory(FORWARD);
135     mydata->moveCounter++;
136     }
137 }
138
139 void checkBETTERcondition() {
140     set_motion(FORWARD);
141 //    NEXT_MOTION = FORWARD;
142     updateMotionHistory(FORWARD);
143     mydata->moveCounter = 0;
144 //    return FORWARD;
145 }
146
147 void checkWORSEcondition() {
148     if (mydata->lastState != WORSE &&
149     mydata->motionHistory[0] == CURRENT_TURN) {
150     changeTurn();
151     }// else if (mydata->lastState == WORSE &&
```

```
152 //         mydata->motionHistory[0] == CURRENT_TURN) {
153 //   setStepTime(QUARTER_TURN);
154 //     } else {
155       setStepTime(TURN_AROUND);
156 //     }
157       set_motion(CURRENT_TURN);
158 //     NEXT_MOTION = CURRENT_TURN;
159       updateMotionHistory(CURRENT_TURN);
160 }
161
162 void stallCollision() {
163 //     if (mydata->waitID == 255) {
164         mydata->waitID = mydata->msgID;
165         mydata->lastTicks = kilo_ticks;
166 //     }
167     if (kilo_uid < mydata->waitID && mydata->waitState == NO_WAIT) {
168         set_motion(FORWARD);
169 //       NEXT_MOTION = FORWARD;
170 //     } else if (kilo_uid > mydata->waitID && mydata->msgWaitState == ...
    FAIL){
171 //   set_motion(FORWARD);
172 //     } else if (mydata->waitState == FAIL && mydata->msgWaitState == ...
    FAIL) {
173 //         setStepTime(TURN_AROUND);
174 //   set_motion(CURRENT_TURN);
175       } else {
176       set_motion(STOP);
177 //   NEXT_MOTION = STOP;
178       }
179 //         mydata->waitState = WAIT;
180 //}
181
182 }
183 //     if (kilo_uid > mydata->msgID) {
184 //         resetStepTime();
185 //   set_motion(STOP);
186 //     } else {
187 //   resetStepTime();
188 //   set_motion(FORWARD);
189 //     }
190
191
192 void stateMachine() {
193     //SMARTER Light Finder (SELF COGNIZANT)
194     switch(mydata->search_state) {
195         case SUCCESS:
196         set_motion(STOP);
197 //      NEXT_MOTION = STOP;
198             updateMotionHistory(STOP);
199         break;
200     case NONE:
201         checkNONEcondition();
202         break;
203         case BETTER:
```

```
204         checkBETTERcondition();
205         break;
206     case WORSE:
207         checkWORSEcondition();
208             break;
209     }
210 }
211
212
213 void loop() {
214
215 //if (mydata->search_state != SUCCESS) {
216     updateMeasurements();
217 mydata->transmit_msg.data[0] = mydata->intensity;// & 0xff;
218 mydata->transmit_msg.data[1] = (mydata->intensity >> 8);
219 mydata->transmit_msg.data[4] = mydata->search_state;
220 //}
221 //ONLY UPDATE MEASUREMENTS IF STEP TIME IS DONE.
222 if (kilo_ticks ≥ mydata->stepTicks + mydata->stepTime) {// &&
223 //    mydata->search_state != SUCCESS) {
224     resetStepTime();
225 //    if (mydata->search_state != SUCCESS) {
226 //        updateMeasurements();
227 //    }
228 //    mydata->lastTicks = kilo_ticks;
229     mydata->stepTicks = kilo_ticks;
230     stateMachine();
231     mydata->lastState = mydata->search_state;
232     if (mydata->intensity == mydata->nBest
233         && mydata->search_state != SUCCESS
234         && kilo_ticks > REDUCTION_TIME) {
235         mydata->ticks_to_success--;
236     }
237   }
238     // VARIABLE BEING TRANSMITTED, CURRENTLY LIGHT INTENSITY.
239 //    mydata->transmit_msg.data[0] = mydata->intensity;//nBest;
240     // Update distance estimate with every message
241     if (mydata->new_message) {
242         mydata->new_message = 0;
243         mydata->cur_distance = estimate_distance(&mydata->dist);
244         if (mydata->msgMeasurement < mydata->nBest) {
245             mydata->nBestID = mydata->msgID;
246         mydata->nBest = mydata->msgMeasurement;
247     }
248     }
249     //TRY TO HANDLE COLLISION
250     if (mydata->cur_distance ≤ TOOCLOSE_DISTANCE && ...
        mydata->search_state != SUCCESS && mydata->msgSuccess != ...
        SUCCESS) {
251     stallCollision();
252     } // IF COLLIDING WITH SUCCESS
253       else if (mydata->cur_distance ≤ TOOCLOSE_DISTANCE && ...
            mydata->search_state != SUCCESS && mydata->msgSuccess == ...
            SUCCESS) {
```

```
254         set_motion(STOP);
255         mydata->search_state = SUCCESS;
256         updateMotionHistory(STOP);
257     }
258 //set_motion(NEXT_MOTION);
259 }
260 
261 
262 void message_rx(message_t *m, distance_measurement_t *d) {
263     mydata->new_message = 1;
264 //    mydata->new_message = m->data[0];
265     mydata->dist = *d;
266     mydata->msgMeasurement = ((uint16_t)m->data[1] << 8) | m->data[0];
267     mydata->msgID = m->data[2];
268     mydata->msgWaitState = m->data[3];
269     mydata->msgSuccess = m->data[4];
270 }
271 
272 void setup_message(void)
273 {
274   mydata->transmit_msg.type = NORMAL;
275 //  mydata->transmit_msg.data[0] = kilo_uid & 0xff; //low byte of ID, ...
      currently not really used for anything
276   // LOW BYTE OF INTENSITY
277   mydata->transmit_msg.data[0] = mydata->intensity;// & 0xff;
278   // HIGH BYTE OF INTENSITY
279   mydata->transmit_msg.data[1] = (mydata->intensity >> 8);
280   mydata->transmit_msg.data[2] = kilo_uid;
281   mydata->transmit_msg.data[3] = mydata->waitState;
282   mydata->transmit_msg.data[4] = mydata->search_state;
283   //finally, calculate a message check sum
284   mydata->transmit_msg.crc = message_crc(&mydata->transmit_msg);
285 }
286 
287 message_t *message_tx()
288 {
289   return &mydata->transmit_msg;
290 }
291 
292 void setup()
293 {
294   mydata->search_state = NONE;
295   mydata->cur_distance = 100;
296   mydata->new_message = 0;
297   mydata->intensity = 1023;
298 
299   mydata->nBest = 1023;//0;
300 //  mydata->nBestID = -1;
301 //  mydata->msgMeasurement = -1;
302 
303   mydata->lastIntensity = 1023;
304   mydata->lastTicks = 0;
305   mydata->stepTicks = 0;
306   mydata->moveCounter = 0;
```

```
307
308   mydata->waitID = 255;
309   mydata->waitState = NO_WAIT;
310   mydata->msgWaitState = NO_WAIT;
311
312   mydata->msgSuccess = NONE;
313   mydata->ticks_to_success = TICKS_TO_SUCCESS;
314
315   setup_message();
316
317   set_color(RGB(0,3,0)); // color of the stationary bot
318 }
319
320
321 #ifdef SIMULATOR
322 /* provide a text string for the simulator status bar about this bot */
323 static char botinfo_buffer[10000];
324 char *cb_botinfo(void)
325 {
326   char *p = botinfo_buffer;
327   p += sprintf (p, "ID: %d \n", kilo_uid);
328 //  p += sprintf (p, "xCoord: %d",);
329 //  p += sprintf (p, "");
330   p += sprintf (p, "Light Intensity: %d \n", mydata->intensity);
331 //  p += sprintf (p, "Succes: %d \n", mydata->search_state);
332 //  p += sprintf (p, "Distance: %d from %d. \n", mydata->cur_distance, ...
      mydata->msgID);
333 //  p += sprintf (p, "Wait ID: %d", mydata->waitID);
334 //  p += sprintf (p, "nBest Intensity: %d \n", mydata->nBest);
335 //  p += sprintf (p, "nBest ID : %d \n", mydata->nBestID);
336 //  p += sprintf (p, "Low byte of intensity : %d \n", mydata->intensity ...
      & 0xff);
337   p += sprintf (p, "rxMsg Intensity: %d \n", mydata->msgMeasurement);
338 //  p += sprintf(p, "Ticks to success : %d \n", mydata->ticks_to_success);
339 //  if (mydata->orbit_state == ORBIT_NORMAL)
340 //    p += sprintf (p, "State: ORBIT_NORMAL\n");
341 //  if (mydata->orbit_state == ORBIT_TOOCLOSE)
342 //    p += sprintf (p, "State: ORBIT_TOOCLOSE\n");
343
344   return botinfo_buffer;
345 }
346 //#endif
347
348 // ATTEMPTED LIGHTING CALLBACK
349 //int16_t cb_lighting(double xCoord, double yCoord)
350 //{
351 //    int16_t intensity = 1023 - sqrt(xCoord*xCoord + yCoord*yCoord);
352 //    int16_t intensity = get_ambientlight(); //BROKEN HERE
353 //    return intensity;
354 //}
355
356 #include <jansson.h>
357 json_t *json_state();
358 #endif
```

```
359
360
361 int main() {
362     kilo_init();
363     kilo_message_rx = message_rx;
364
365
366     // bot 0 is stationary and transmits messages. Other bots orbit ...
          around it.
367 //    if (kilo_uid == 0)
368     kilo_message_tx = message_tx;
369
370     kilo_start(setup, loop);
371
372     SET_CALLBACK(botinfo, cb_botinfo);
373 //    SET_CALLBACK(lighting, cb_lighting);
374     SET_CALLBACK(json_state, json_state);
375
376     return 0;
377 }
```

D.5 Rastrigin CS-CPVM

This is functionally the same as CS-CPVM except the lighting callback is manually defined.

bestLightFinder.c

```
348 // LIGHTING CALLBACK DEFINED HERE
349 int16_t cb_lighting(double xCoord, double yCoord)
350 {
351     double x = xCoord/100;
352     double y = yCoord/100;
353 //    int16_t calc_intensity = pow(x,2) - 10*cos(M_PI/2*x);
354     double calc_intensity = pow(x,2) - 10*cos(M_PI/2*x);
355     calc_intensity += pow(y,2) - 10*cos(M_PI/2*y);
356     calc_intensity += 20;
357 //    calc_intensity = calc_intensity; * 10;
358     int16_t ret_intensity = (int16_t) calc_intensity;
359 //    printf("Intensity: %d\n", ret_intensity);
360     return ret_intensity;
361 }
```

json_state.c

```c
/* Saving bot state as JSON. */

#include <kilombo.h>

#ifdef SIMULATOR

#include <jansson.h>
#include <stdio.h>
#include <string.h>
#include "orbit.h"

json_t *json_state() {
    json_t* state = json_object();

    json_t* li = json_integer(mydata->intensity);
    json_object_set(state, "Light Intensity", li);
    json_t* ss = json_integer(mydata->search_state);
    json_object_set(state, "Search State", ss);

    return state;
}

#endif
```