FSU PANAMA CITY

EEL 4710 Introduction to VHDL

Performed by

Piper Ellsworth Jaehyun Lee Shawn Hagler

200843170 200820584 200697142

Instructor: Dr. Manzak

Summer 2023

Flower Pollination Algorithm

Abstract

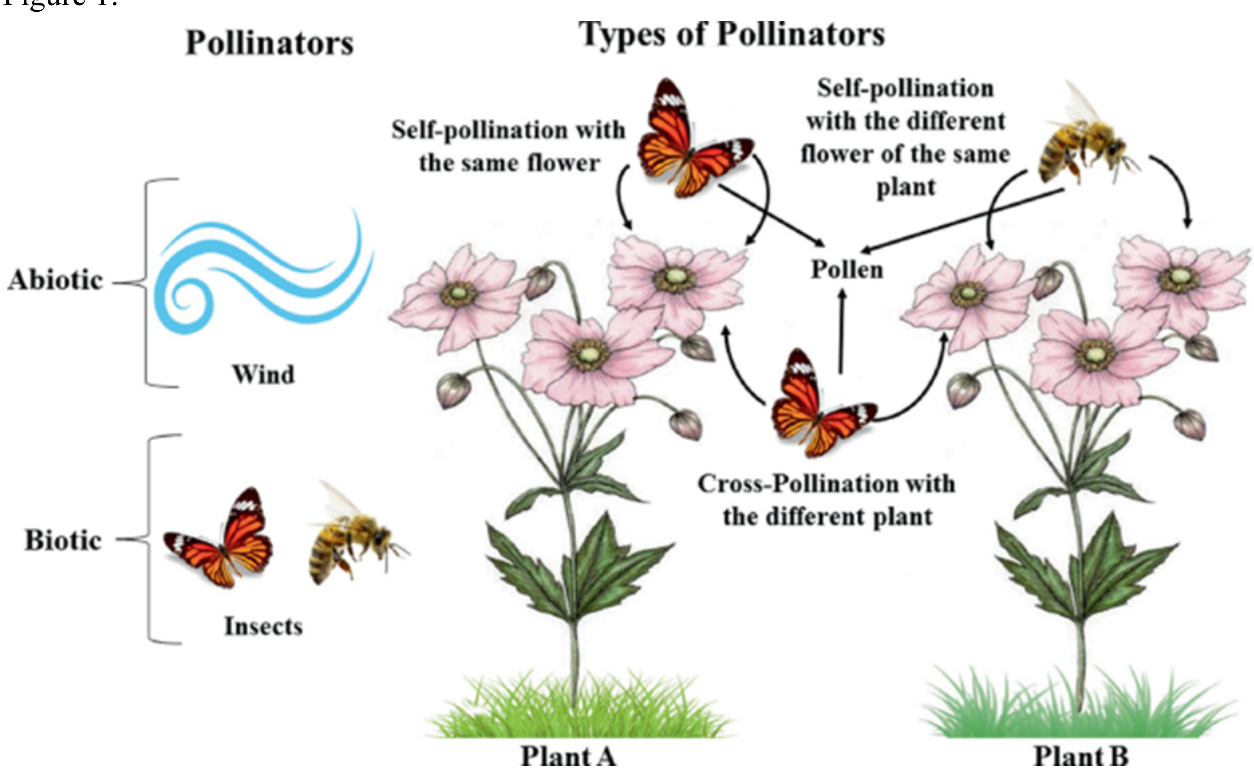
The Flower Pollination Algorithm (FPA) is a nature-inspired optimization algorithm that replicates the global and local movements of pollinating bees and flowers in nature. In this project, we examine and analyze the potential applications, advantages, and limitations of the FPA. Notably, the FPA is illustrated to have high efficiency and is particularly potent in solving complex optimization problems. The paper also offers a deeper exploration of underlying mathematical equations, design, and function of the FPA. Lastly, outline the challenges faced during the implementation of FPA within an FPGA-based system, and discuss possible solutions to address them.

Introduction

Big Picture

The Flower Pollination Algorithm (FPA) is based on the natural process of flower pollination. Figure 1 gives a visual example of pollination. Just like in nature, there are two types of "pollination" in the FPA - global and local. Cross-pollination is when bees or butterflies carry pollen far away from flowers on other plants. In the algorithm, this is represented as a global optimum and it occurs when a completely new solution is tried for, and is not too close to what we have already. Local optimum is like a flower pollinating itself or with a close neighbor, it's a small change or adjustment to the current solution. The algorithm keeps doing this, making small changes sometimes (local pollination) and big jumps to new areas at other times (global pollination). Just as pollination propagates the fittest genes, this process keeps the best solution (the current minimum) found so far and discards the rest. The algorithm keeps doing this until it either finds the best solution (minimum) like a bee finding the best flower, or until a pre-set amount of time passes. This is how the Flower Pollination Algorithm turns the natural process of flower pollination into a problem-solving tool!

Figure 1:

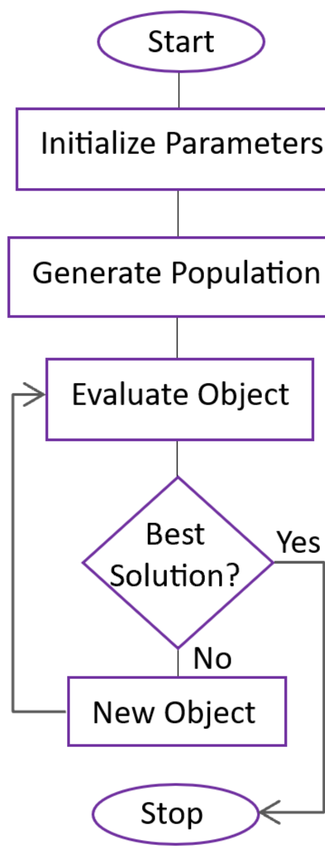


Procedure and Math

Procedure

This algorithm is all about finding either the maximum or minimum of your data set. To do that, the algorithm compares each item to the previous max or min. The algorithm starts by initializing the parameters of the search. A simple diagram detailing the following steps can be seen in Figure 2. This would be deciding between using the global or local and the maximum or minimum. Next, the algorithm will generate the population based on your data. In other words, it will use the function being used in the algorithm to find the various data points. It will then randomly pick one of the data points (or object) and automatically make the value the maximum or minimum based on what the parameters. Assume maximum for this explanation. It will then compare it to nothing for the first iteration which automatically makes it greater, so another object is selected and compared to the max object. If the new object is greater than the current max object, it becomes the max object. The process repeats until every object or point has been checked and compared. A deeper explanation of the more intricate details can be found in the existing works section along with an in-depth flow chart in Figure 10.

Figure 2:



Mathematics

There are several equations that go into making an optimization algorithm work. The Flower Pollination Algorithm has several equations for both the cases of global and local pollination. Most papers found on FPA follow the same structure, but they all use different variables. The main equation for global pollination is eq. 1 below.

$$x_i^{t+1} = x_i^t + L(x_i^t - g*) \quad \text{[eq. 1]}$$

The variables mean the subscript i represents the flower, and t is the iteration. x_i^t represents the actual desired resultant vector. g* represents the current best solution at iteration t [1]. The L in eq.1 refers to the Levy Flight Distribution which is seen in equation (eq.2) seen below. The variable β here represents the gamma function which is a random number generator.

$$L \sim \frac{(\beta + 1) \times sin\left(\frac{\beta\pi}{2}\right)}{\pi} \times \frac{1}{s^{\beta+1}}, \quad (s \gg s_o > 0)$$

[eq.2]

For local pollination, equation 3 below is used. Here the variables x_j^t and x_k^t represent two different flowers from within the same population.

$$x_i^{t+1} = x_i^t + \epsilon\left(x_j^t - x_k^t\right)$$

[eq.3]

For both cases, the variable for the flower is calculated using the following equation set (eq.4).

$$x_i^{t+1} = \begin{pmatrix} x_i^{t+1}, & if\ j\left(x_i^{t+1}\right) < j\left(x_i^t\right) \\ x_i^t, & else\ j\left(x_i^{t+1}\right) \geq j\left(x_i^t\right) \end{pmatrix}$$

[eq.4]

Equation 1 is the equation in the algorithm used to determine the value of the global minimum. Within equation 1 equations 2 and 4 are applied. Equation 2 calculates the Levy Flight value, and equations 4 calculates the variable for the flower. Equation 3 is the equation for the local minimum. It also utilizes equation 4 within it.

Advantages and Disadvantages

Like anything, the Flower Pollination Algorithm has its ups and downs. It just so happens to have more advantages which made it a great candidate for this project. To combat the complex nature of optimization problems, FPA uses parameter tuning. This allows the algorithm to maximize performance and minimize loss which allows for more efficiency. It also has the fastest and most accurate optimization algorithm for optimal parameter extraction [2]. This is when the algorithm finds parameters so that the simulation and actual measured value are very similar. Another benefit is the exponentially fast convergence rate. Flower Pollination Algorithm converges quicker than other algorithms to either the maximum or the minimum. However, this does cause the algorithm to converge prematurely and settle on the wrong data point. Based on tests done by Xin-She Yang, FPA is more efficient than two of the most popular metaheuristic optimization algorithms currently in use. Another issue lies in the "lack of perfect compromise between global exploration and local exploitation" [3]. This is referring to the tradeoff known as the exploration-exploitation-dilemma. However, this is a common problem between many similar algorithms. The issue is deciding when the algorithm should conclude its search. Should it continue to explore every piece of data and come to the exact optimal solution but take longer to arrive there? Or should it go to the quicker solution based on what is currently known and possibly miss the most optimal? Researchers are currently working on solving this issue.

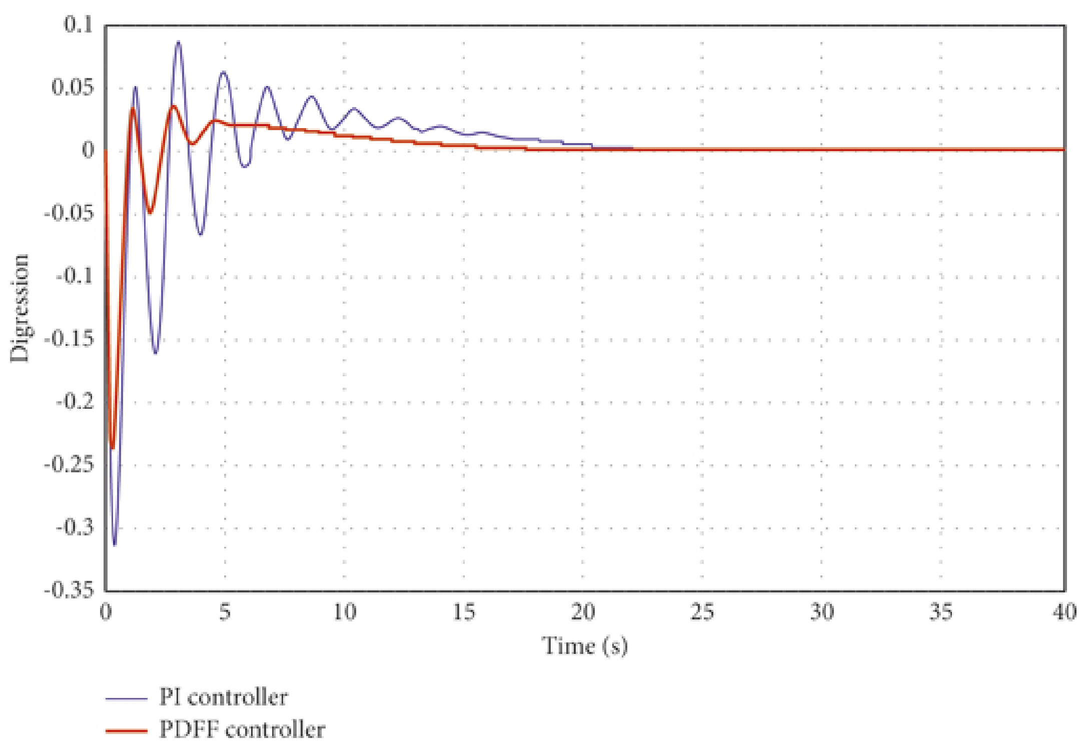
Applications

Flower Pollination Algorithm has many different types of applications. One of the big ones is signal and image processing. The algorithm can easily find global and local maximums and minimums in a long signal. Other applications include computer gaming and wireless sensor networking. Another big application of FPA is power systems. It is very useful in determining the best scenario for specific loads.

Real World Application

A group of engineers at the University of KwaZulu-Natal, used the Flower Pollination Algorithm on their project to optimize power flow. Their goal was to achieve automatic generation control in their power system. Automatic generation control is when a system adjusts the power output of multiple generators at different power plants, in response to changes in the load [4]. Originally the engineers were using a Proportional Integral (PI) controller, but they wanted to upgrade which made them switch to a Pseudo Derivative Feedforward with Feedback (PDFF) controller. They plan to implement FPA into the PDFF controller. This allows for the optimal dynamic performance to be found for each different type of power flow in a reconstructed power system. Figure 3 shows the results of their simulation comparing the power system before and after implementing FPA into the PDFF controller. Both curves converge to the same point of zero, but the one using FPA fluctuates much less and narrows in on the point much sooner than the other method [4].

Figure 3:



Existing Example

First Design

The Flower Pollination Algorithm (FPA), proposed by Xin-She Yang in 2012, is based on the characteristics of flower pollination, such as global and local pollination processes, flower constancy, and reproduction probability. It introduces the inspiration for FPA, explains the algorithm details, presents numerical experiments, and compares FPA's performance with other established optimization algorithms. They identified four key components of flower pollination: biotic and abiotic pollination, global and local pollination, and switch probability [5]. The researchers modeled the global and local pollination mathematically and combined them to form the FPA. The paper then demonstrates the efficacy of FPA through numerical experiments. The researchers tested FPA against four benchmark functions widely used in global optimization and compared the results to other optimization algorithms: Genetic Algorithm (GA), Particle Swarm Optimization (PSO), and Bat Algorithm. FPA consistently performed well and outperformed other algorithms in certain cases. Furthermore, the researchers also found that FPA could solve complex multidimensional problems very efficiently [5].

Significant Scientific Papers

Flower Pollination Algorithm Parameters Tuning

The researchers conducted extensive computational experiments to validate the FPA using the flowchart shown in Figure 10 [1]. They used a set of standard benchmark functions to test the performance of their tuned FPA and compared it with the standard FPA and other well-known optimization algorithms. The results demonstrated that the adaptive tuning of FPA parameters significantly improved the algorithm's performance in terms of convergence speed, solution quality, and robustness. It delivered superior or comparable results to other algorithms on benchmark functions. Figure 4 shows the benchmark functions that are being tested throughout the paper [1].

Figure 4:

	Function No	Function Name	Global optimum f_i^*
Unimodal	1	Sphere Function	– 1400
	2	Rotated High Conditioned Elliptic Function	– 1300
	3	Rotated Bent Cigar Function	– 1200
	4	Rotated Discus Function	– 1100
	5	Different Powers Function	– 1000
Basic Multimodal	6	Rotated Rosenbrock's Function	– 900
	7	Rotated Schaffers F7 Function	– 800
	8	Rotated Ackley's Function	– 700
	9	Rotated Weierstrass Function	– 600
	10	Rotated Griewank's Function	– 500
	11	Rastrigin's Function	– 400
	12	Rotated Rastrigin's Function	– 300
	13	Non-Continuous Rotated Rastrigin's Function	– 200
	14	Schwefel's Function	– 100
	15	Rotated Schwefel's Function	100
	16	Rotated Katsuura Function	200
	17	Lunacek Bi_Rastrigin Function	300
	18	Rotated Lunacek Bi_Rastrigin Function	400
	19	Expanded Griewank's plus Rosenbrock's Function	500
	20	Expanded Scaffer's F6 Function	600
Composite Multimodal	21	Composition Function 1	700
	22	Composition Function 2	800
	23	Composition Function 3	900
	24	Composition Function 4	1000
	25	Composition Function 5	1100
	26	Composition Function 6	1200
	27	Composition Function 7	1300
	28	Composition Function 8	1400

Flower Pollination Algorithm for Solving Constrained Optimization Problems

Another significant paper published by Gandomi et al. in 2013, presents a novel optimization algorithm model called the Flower Pollination Algorithm (FPA), which was developed to solve constrained optimization problems. Inspired by the natural pollination process of flowering plants, the algorithm uses a mix of global and local search approaches to achieve optimization. The paper includes both theoretical discussions on FPA and application-based evidence of its effectiveness, showcasing its successful implementation in solving various numerical and engineering problems. Based on the outcomes, the authors claim that the FPA outperforms several established optimization algorithms [10].

A Novel Method Motivated from the Behavior of Flowers for Optimal Solution

Lastly, in 2020 Decoderz analyzed a method for optimal solution based on the behavior of flowers, named the Flower Pollination Algorithm (FPA). The article elaborates on how the algorithm works, providing a detailed analysis of its steps and methodologies. The author makes a convincing argument for the effectiveness of the FPA, rooting for its implementation in a range of optimization problem scenarios. The article also mentions the potential benefits and application areas of FPA, eventually concluding that this innovative algorithm could provide optimal solutions in a variety of contexts [2].

Significant Application

Due to the limitations and complexity of the FPA algorithm, it was not possible to find and sort of hardware implementation of the algorithm. Also, the algorithm is mainly for optimization, which made it harder to implement onto hardware. However, the algorithm was used in different applications in order to experiment and solve different problems.

Experimental Implementation of Flower Pollination Algorithm for Speed Controller of a BLDC Motor

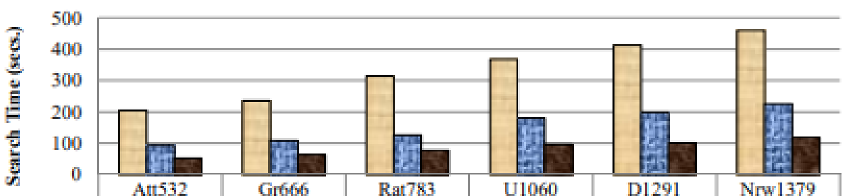
This paper presents an experimental implementation of the Flower Pollination Algorithm (FPA) for speed control of a Brushless Direct Current (BLDC) motor. The researchers utilized the algorithm to optimize the Proportional Integral Derivative (PID) controller parameters for the speed control system of the motor. Using the FPA, it aimed to find the optimal PID parameters (proportional gain, integral gain, derivative gain) that would minimize the overall error in the system and increase efficiency. Also, implemented the optimized PID controller in a real BLDC motor speed control system. The results showed significant improvement in the motor's performance, specifically in terms of settling time, overshoot, and steady-state error [6].

Optimal Solving Large-Scale Traveling Transportation Problems by Flower Pollination Algorithm

Another application found explores the effectiveness of the Flower Pollination Algorithm (FPA) in solving large-scale Traveling Transportation Problems (TTPs). The TTP, often referred to as the Traveling Salesman Problem (TSP) in transportation literature, involves determining the shortest possible route that a traveling entity (like a salesman or a vehicle) can take to visit a set of destinations and return to the origin, thereby saving on time and cost. The researchers implemented a computer model to apply the FPA to a set of large-scale TTP benchmarks. They compared the performance of the FPA with established heuristics, such as the Genetic Algorithm (GA), Particle Swarm Optimization (PSO), and Tabu Search (TS). As shown in Figure 5 and Figure 6, FPA showed a fairly significant performance compared to other heuristics on the benchmark problems, demonstrating its efficiency in solving TTP, especially on a large scale. The algorithm was found to be effective in finding near-optimal or optimal solutions with relatively low computational time, affirming the suitability of FPA for large-scale complex optimization problems [7].

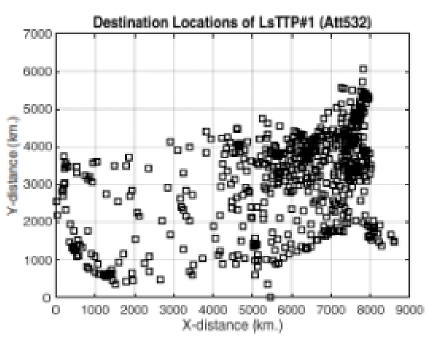
Figure 5:

Entries	Names	Optimal Solutions (Km.)	GA (Km.)	PSO (Km.)	FPA (Km.)
LsTTP#1	Att532	92,794	96,858.78	94,490.88	**92,797.62**
LsTTP#2	Gr666	294,358	312,754.15	301,218.84	**295,018.46**
LsTTP#3	Rat783	8,806	9,548.63	9,014.56	**8,810.71**
LsTTP#4	U1060	224,094	238,474.95	234,454.38	**225,174.58**
LsTTP#5	D1291	50,801	53,748.28	51,464.87	**50,866.23**
LsTTP#6	Nrw1379	56,638	60,101.47	58,415.83	**56,767.02**

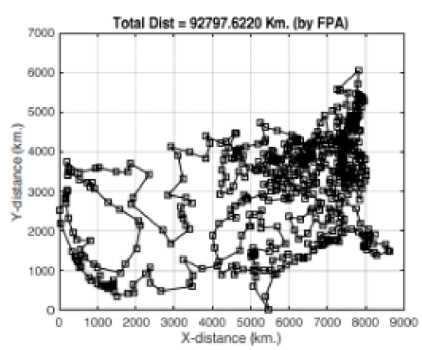


	Att532	Gr666	Rat783	U1060	D1291	Nrw1379
GA	205.17	236.36	314.85	367.74	412.39	459.89
PSO	93.36	108.42	126.28	179.91	199.57	225.34
FPA	52.59	63.45	78.13	96.37	101.42	118.54

Figure 6:



Destination locations of LsTTP#1 (Att532).



Optimal tour of LsTTP#1 by FPA.

Uniqueness of FPGA Implementation

One key benefit is the ability to execute tasks in parallel, allowing for faster execution of complex optimization problems or real-time applications. Additionally, FPGA can be customized and reprogrammed to meet specific task requirements, resulting in improved efficiency and performance. The direct data pathway in FPGA hardware reduces latency compared to software-based implementations or generic hardware platforms. FPGAs are also more power-efficient than GPUs or high-performance CPUs, making them ideal for applications where power consumption is a concern. Furthermore, FPGAs have high reliability and stability, with a high tolerance for radiation and a lower risk of single-point failures, making them suitable for critical applications that require continuous operation.

Example

In the Python code, we plan to write this in VHDL for the final report and presentation, provided in the Appendix, the Flower Pollination Algorithm (FPA) is implemented to solve the six-hump camelback function. This is a standard benchmark optimization problem in mathematics and computer science, characterized by its many local minima and two global minima points. This function is defined over a two-dimensional domain, and it is described as follows (eq.5):

$$f(x, y) = (4 - 2.1x^2 + (x^4/3))x^2 + xy + 4(y^2 - 1)y^2 \quad \text{[eq. 5]}$$

The aim is to find the global minimum point of this function, which is represented as x = [-0.0898, 0.0898], y = [0.7126, -0.7126] and the minimum function value should be -1.0316. Notably, in complex optimization scenarios like this, traditional methods can get trapped in local minima positions; hence, the use of the FPA, is a method inspired by the flower pollination process.

In the Python script provided, the FPA starts by initializing a population of random solution vectors (referred to as 'flowers'). For each iteration, the algorithm performs a global pollination (Levy flights, which allow for large-scale explorations of the solution space) or a local pollination (small, random walks for local exploitation). A plot is then rendered, showing the convergence of the algorithm over time to the global minima solution. It should be noted that due to the stochastic nature of the algorithm, multiple runs may give slightly different results, but on average, the algorithm should converge toward the global minimum. The script uses a population size of 175 flowers and runs for 300 iterations. The parameters gamma (the step size), lamb (the scaling factor), and p (the switching probability between global and local pollination) have been chosen empirically. Adjusting these parameters can affect the algorithm's performance and convergence rate.

Figure 7: Python Code

```
import random
import math
import os
import time

def initial_position(flowers, min_values, max_values):
    # initialize a matrix with zero values
    position = [[0] * (len(min_values) + 1) for _ in range(flowers)]

    # iterate through each flower
    for i in range(0, flowers):
        # generate a random position for each x and y coordinate
        for j in range(0, len(min_values)):
            position[i][j] = random.uniform(min_values[j], max_values[j])

        # set the last column value as the evaluation of the six hump
        # camel back function at the position
        position[i][-1] = six_hump_camel_back(position[i][0:len(min_values)])

    # return the matrix of initial flower positions
    return position

def levy_flight(beta):
    # generate two random numbers
    r1 = int.from_bytes(os.urandom(8), byteorder = "big") / ((1 << 64) - 1)
    r2 = int.from_bytes(os.urandom(8), byteorder = "big") / ((1 << 64) - 1)

    # calculate the sigma numerator
    sig_num = math.gamma(1 + beta) * math.sin((math.pi * beta) / 2.0)

    # calculate the sigma denominator
    sig_den = math.gamma((1 + beta) / 2) * beta * 2**((beta - 1) / 2)

    # calculate the sigma value
    sigma = (sig_num / sig_den)**(1 / beta)

    # calculate the levy step length and return the value
    levy = (0.01 * r1 * sigma) / (abs(r2)**(1 / beta))
    return levy

def clip(num, min_value, max_value):
    return max(min(num, max_value), min_value)
```

```python
def pollination_global(position, best_global, flower, gama, lamb,
                       min_values, max_values):
    # create a copy of the best global position
    x = list(best_global)

    # update the x and y coordinates of the position using global pollination
    for j in range(0, len(min_values)):
        x[j] = clip((position[flower][j]  + gama * levy_flight(lamb) *
                     (position[flower][j] - best_global[j])),
                    min_values[j], max_values[j])

    # set the last column value as the evaluation of the six hump
    # camel back function at the position
    x[-1]  = six_hump_camel_back(x[0:len(min_values)])

    # return the new position
    return x

def pollination_local(position, best_global, flower, nb_flower_1, nb_flower_2,
                      min_values, max_values):
    # create a copy of the best global position
    x = list(best_global)

    # update the x and y coordinates of the position using local pollination
    for j in range(0, len(min_values)):
        # generate a random number
        r = int.from_bytes(os.urandom(8), byteorder = "big") / ((1 << 64) - 1)
        x[j] = clip((position[flower][j]  + r *
                     (position[nb_flower_1][j] - position[nb_flower_2][j])),
                    min_values[j], max_values[j])

    # set the last column value as the evaluation of the six hump
    # camel back function at the position
    x[-1] = six_hump_camel_back(x[0:len(min_values)])

    # return the new position
    return x
```

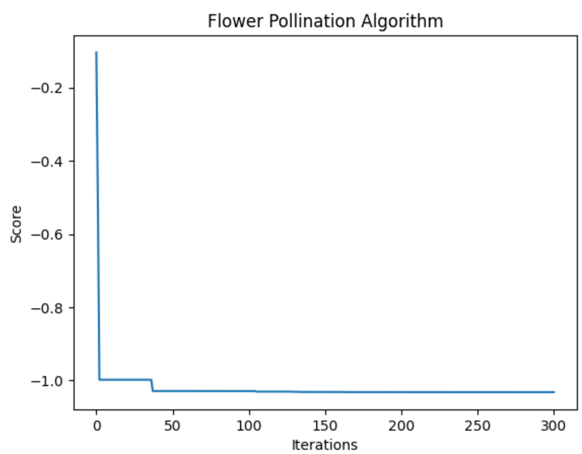
```python
def fpa(flowers, min_values, max_values, iterations, gama, lamb, p):
    # record the start time of the algorithm
    start = time.time()

    # initialize the positions of the flowers
    position = initial_position(flowers, min_values, max_values)

    # find the best global position from the initial flowers
    best_global = sorted(position, key=lambda x: x[-1])[0]

    # create a copy of the best global position
    x = list(best_global)

    # iterate through the set amount of iterations
    for count in range(iterations):
        # print the current iteration and the best position found
        print("Iteration = ", count, " f(x) = ", best_global[-1])

        # iterate through each flower
        for i in range(0, len(position)):
            # choose two random flowers for local pollination
            nb_flower_1 = int(random.random() * len(position))
            nb_flower_2 = int(random.random() * len(position))

            # ensure that the two flowers are not the same
            while nb_flower_1 == nb_flower_2:
                nb_flower_1 = int(random.random() * len(position))

            # generate a random number between 0 and 1
            r = int.from_bytes(os.urandom(8), byteorder = "big") / ((1 << 64) - 1)

            # if the random number is less than p then perform global pollination
            # otherwise perform local pollination
            if (r < p):
                x = pollination_global(position, best_global, i, gama, lamb,
                                       min_values, max_values)
            else:
                x = pollination_local(position, best_global, i, nb_flower_1,
                                      nb_flower_2, min_values, max_values)

            # if the new position results in a better solution, then
```

```python
            # if the new position results in a better solution, then
            # update the current position
            if (x[-1] <= position[i][-1]):
                for j in range(0, len(x)):
                    position[i][j] = x[j]

            # if the best position has been improved then update it
            value = sorted(position, key=lambda x: x[-1])[0]
            if (best_global[-1] > value[-1]):
                best_global = list(value)

    # record the end time of the algorithm
    end = time.time()
    return best_global

def six_hump_camel_back(variables_values):
    return 4 * variables_values[0]**2 - 2.1 * variables_values[0]**4 + (1/3) * variables_values[0]**6 + \
        variables_values[0] * variables_values[1] - 4 * variables_values[1]**2 + 4 * variables_values[1]**4

best_solution = fpa(175, [-5,-5], [5,5], 300, 0.1, 1.5, 0.8)
```

Figure 8:

Iteration	f(x)
0	-0.8708680541
10	-1.011545046
40	-1.011547615
70	-1.017303535
80	-1.031034176
110	1.031517444
170	-1.031548803
175	-1.031578976
200	-1.031591313
250	-1.031608242
260	-1.031609412
270	-1.031622394
300	-1.031626743

Figure 9:



Methodology

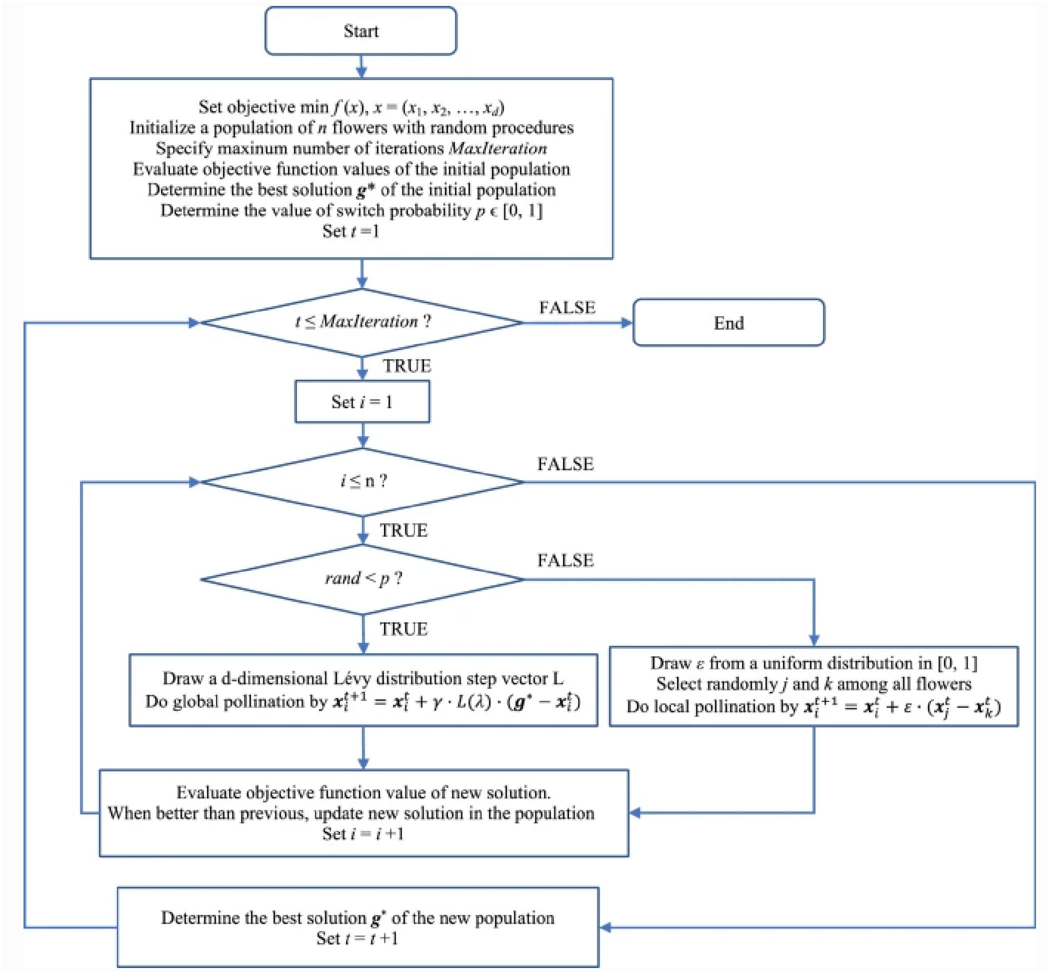
VHDL Code Division

Coding for the project was mainly divided into three parts, developing, updating, and testing. Developing the structure and functions for the project, updating the code to execute and output the expected results, then testing and debugging to test different inputs and variables.

Data Flow Graph

Figure 10 is a data flow chart proposed by Xin-She Yang [1]. This chart was used to generate the original Python code for this algorithm as well as the modified Python code and VHDL code were created during this project.

Figure 10:



Start
Set objective min $f(x)$, $x = (x_1, x_2, \ldots, x_d)$
Initialize a population of n flowers with random procedures
Specify maxinum number of iterations *MaxIteration*
Evaluate objective function values of the initial population
Determine the best solution g^* of the initial population
Determine the value of switch probability $p \in [0, 1]$
Set t =1
$t \leq MaxIteration$?
FALSE
End
TRUE
Set i = 1
$i \leq$ n ?
FALSE
TRUE
$rand < p$?
FALSE
TRUE
Draw a d-dimensional Lévy distribution step vector L
Do global pollination by $x_i^{t+1} = x_i^t + \gamma \cdot L(\lambda) \cdot (g^* - x_i^t)$
Draw ε from a uniform distribution in [0, 1]
Select randomly j and k among all flowers
Do local pollination by $x_i^{t+1} = x_i^t + \varepsilon \cdot (x_j^t - x_k^t)$
Evaluate objective function value of new solution.
When better than previous, update new solution in the population
Set $i = i$ +1
Determine the best solution g^* of the new population
Set $t = t$ +1

Coding Alternative

The Python code from the example section was rewritten into VHDL and modified to generate an alternative coding solution for the project.

Figure 11: VHDL

```vhdl

package flower_pollination_package is
    -- type that holds a dymanic amount of real numbers to represent a vector
    type real_vector is array(natural range <>) of real;
    -- type that holds a dynamic amount of real_vector(1 to 3) to represent a matrix
    type real_vector_vector is array(natural range <>) of real_vector(1 to 3);
end flower_pollination_package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use work.flower_pollination_package.all;

entity flower_pollination is
    port (
        clk: in std_logic;                          -- clock signal
        reset: in std_logic;                        -- reset signal
        flower_count: in integer;                   -- numnber of flowers
        min: in integer;                            -- min value
        max: in integer;                            -- max value
        gamma: in real;                             -- gamma value
        lamb: in real;                              -- lambda value
        p: in real;                                 -- probability value
        iterations: in integer;                     -- number of iterations
        best_solution: out real_vector(1 to 3) -- best solution output
    );
end flower_pollination;

architecture beh of flower_pollination is
    -- generates two random real numbers
    function random_real return real_vector is
            variable seed1: integer := 123456789;
            variable seed2: integer := 987654321;
            variable seed3: integer := 239438458;
            variable rand_real1: real;
            variable rand_real2: real;
            variable random: real_vector(1 to 2);
    begin
        -- generate the first random real number
        uniform(seed1, seed2, rand_real1);
        -- generate the second random real number
        uniform(seed2, seed3, rand_real2);
        random(1) := rand_real1;
        random(2) := rand_real2;
        return random;
    end random_real;

    -- generate a random integer bounded by the min and max values
    function random_integer(min_value: integer; max_value: integer) return integer is
        variable real_random: real;
        variable seed1, seed2: positive := 987654321;
    begin
        -- generate the random real number
        uniform(seed1, seed2, real_random);
        -- bound the number and convert it to an integer
        return integer(real(min_value) + real_random * real(max_value - min_value + 1));
    end random_integer;

    -- Six Hump Camel Back objective function
    function six_hump_camel_back(variables: real_vector) return real is
            variable x: real := variables(1);
            variable y: real := variables(2);
    begin
        return 4.0 * (x**2) - 2.1 * (x**4) + ((1.0/3.0) * (x**6)) + x * y - 4.0 * (y**2) + 4.0 * (y**4);
    end six_hump_camel_back;
```

```
-- initialize the flower's positions
function initial_positions(flower_count: integer; min: integer; max: integer) return real_vector_vector is
    variable positions: real_vector_vector(1 to flower_count);
begin
    -- iterate through each flower
    for i in 1 to flower_count - 1 loop
        -- generate a random coordinate for each x and y value within the specified search space
        for j in 1 to 2 loop
            positions(i)(j) := real(min) + random_real(j) * real(max - min + 1);
        end loop;
        -- evaluate the fitness of the position
        positions(i)(3) := six_hump_camel_back(positions(i));
    end loop;
    return positions;
end initial_positions;

-- calculate the step length using Levy flight to determine how far a solution moves
function levy_flight(beta: real) return real is
    constant sig_num: real := 0.9399856029866254;
    constant sig_den: real := 1.6168504121556964;
    variable sigma: real;
    variable levy: real;
    variable r1, r2: real;
begin
    r1 := random_real(1);
    r2 := random_real(1);
    sigma := (sig_num / sig_den) ** (1.0 / beta);
    levy := (0.01 * r1 * sigma) / (real(abs(r2)) ** (1.0 / beta));
    return levy;
end levy_flight;

-- performs global pollination where a flower can pollinate with any flower in the environment
function pollination_global(positions: real_vector_vector; best_position: real_vector; min: integer;
                            max: integer; flower: integer; gamma: real; lamb: real) return real_vector is
    variable x: real_vector(1 to 3);
    variable delta: real;
begin
    -- create a new x and y coordinate position using global pollination
    for i in 1 to 2 loop
        x(i) := positions(flower)(i) + gamma * levy_flight(lamb) * (best_position(i) - positions(flower)(i));
        -- ensure the value is within boundaries
        if x(i) < real(min) then
            x(i) := real(min);
        elsif x(i) > real(max) then
            x(i) := real(max);
        end if;
    end loop;
    -- evaluate the fitness of the position
    x(3) := six_hump_camel_back(x);
    return x;
end pollination_global;

-- performs local pollination where a flower can only pollinate with its neighboring flowers
function pollination_local(flower_count: integer; flower: integer; positions: real_vector_vector;
                           min: integer; max: integer) return real_vector is
    variable x: real_vector(1 to 3);
    variable delta: real;
    variable r: real;
    variable nb_flower_1: integer := random_integer(1, flower_count);
    variable nb_flower_2: integer := random_integer(1, flower_count);
begin
    r := random_real(1);
    -- create a new x and y coordinate position using local pollination
    for i in 1 to 2 loop
        delta := r * (positions(nb_flower_1)(i) - positions(nb_flower_2)(i));
        -- ensure the value is within boundaries
        if (positions(flower)(i) + delta) > real(max) then
            x(i) := real(max);
        elsif (positions(flower)(i) + delta) < real(min) then
            x(i) := real(min);
        else
            x(i) := positions(flower)(i) + delta;
        end if;
    end loop;
    -- evaluate the fitness of the position
    x(3) := six_hump_camel_back(x);
    return x;
end pollination_local;
```

```

    signal count: integer := 0;
begin
    process (clk, reset)
        variable positions: real_vector_vector(1 to 175);
        variable best_position: real_vector(1 to 3);
        variable x: real_vector(1 to 3);
    begin
        if (reset = '1') then
            -- reset the count, initialize the flower positions, and set the default best position
            count <= 0;
            positions := initial_positions(flower_count, min, max);
            best_position := positions(1);
        elsif (rising_edge(clk)) then
            -- keep running the algorithm for the specified number of iterations
            if (count < iterations) then
                -- iterate through each flower
                for i in 1 to flower_count loop
                    -- if a random number is less than p then perform global pollination
                    -- otherwise perform local pollination
                    if (random_real(1) < p) then
                        x := pollination_global(positions, best_position, min, max, i, gamma, lamb);
                    else
                        x := pollination_local(flower_count, i, positions, min, max);
                    end if;
                    -- compare the new position to the best position and update if better
                    if (fitness_compare(positions(i), best_position)) then
                        best_position := positions(i);
                    end if;
                end loop;
                -- increment the count
                count <= count + 1;
            end if;
        else
            -- set the output best solution to the best position found
            best_solution <= best_position;
        end if;
    end process;
end beh;
```

Creative Solution

In order to output the expected results such as the simulation waveform showing different minimum and variable values every iteration, it was required to generate different random values every iteration of the algorithm. The solution that was utilized in the project is changing the random value every clock cycle, so each iteration will be calculating the output using different random values.

Limitations

Despite the noted advantages, there are a few limitations in the FPA algorithm. One notable limitation is the challenge of premature convergence. The algorithm tends to find a solution quickly but this solution is often not the best possible solution since the algorithm settles prematurely. Furthermore, the FPA algorithm only replicates the strategies found in natural pollination and therefore lacks the ability to adapt to changing conditions, this leading to less optimal solutions in certain scenarios. Another significant limitation exists in the computational resources required, as complex problems require larger population sizes and higher numbers of iterations, leading to potentially high computational costs.

Results and Appendices

Figure 12: VHDL Code

```
package flower_pollination_package is
 -- type for a vector with real values
    type real_vector is array(natural range <>) of real;
end flower_pollination_package;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.math_real.all;
use work.flower_pollination_package.all;

entity final2 is
  Port ( clk, rst : in std_logic;   -- clock and reset signal
        flower_count: in integer;   -- number of flowers
        iteration : in integer;     -- number of iterations
        min, max: in integer;       -- minimum and maximum boundary
        gamma, lamb: in real;       -- gamma and lmbda value
        p: in real;                 -- probability value
        xxx : in real;              -- random real value
         -- vector with random real value for functions and equations
        x_vector,levy_vector, local_vector : in real_vector(1 to 2);
        best_x : out real_vector(1 to 2); -- vector of best values
        fx : out real );            -- best solution(minimum)
end final2;

architecture Behavioral of final2 is
    -- function to calculate the levy flight value
    function levy_flight(beta: real; levy_vector: real_vector) return real is
        constant sig_num: real := 0.9399856029866254;
        constant sig_den: real := 1.6168504121556964;
        variable sigma, levy: real;
        variable r1, r2: real;
    begin
        r1 := levy_vector(1);
        r2 := levy_vector(2);
        sigma := (sig_num / sig_den) ** (1.0 / beta);
        levy := (0.01 * r1 * sigma) / (real(abs(r2)) ** (1.0 / beta));
        return levy;
    end levy_flight;

```

```
    -- function to calculate the global pollination value
    function pollination_global(position: real_vector; levy_vector: real_vector; best_position: real_vector;
                                    min: integer; max: integer; gamma: real; lamb: real) return real_vector is
        variable x: real_vector(1 to 2);
        variable delta: real;
    begin
        -- calculate each coordinate positions
        for i in 1 to 2 loop
            x(i) := position(i) + gamma * levy_flight(lamb, levy_vector) * (best_position(i) - position(i));
            -- make sure the positions are within the boundaries
            -- otherwise set to min or max
            if x(i) < real(min) then
                x(i) := real(min);
            elsif x(i) > real(max) then
                x(i) := real(max);
            end if;
        end loop;
        return x;
    end pollination_global;

    -- function to calculate the local pollination value
    function pollination_local(local_vector: real_vector; position: real_vector; xxx: real; min: integer;
        variable x: real_vector(1 to 2);
        variable delta: real;
        variable r: real;
     begin
        r := xxx; -- random real value
        -- find the delta value to calculate each coordinate positions
        for i in 1 to 2 loop
            delta := r * (position(i) - local_vector(i));
            -- make sure the positions are within the boundaries
            -- otherwise set to min or max
            if (position(i) + delta) > real(max) then
                x(i) := real(max);
            elsif (position(i) + delta) < real(min) then
                x(i) := real(min);
            else
                x(i) := position(i) + delta;
            end if;
        end loop;
        return x;
    end pollination_local;

begin
    process (clk, rst)
        variable xx : real_vector(1 to 2);
        variable temp, minimum : real := 0.0;
        variable best_vector : real_vector(1 to 2) := (others => 0.0);
        variable count : integer;
    begin
        if (rst = '1') then -- reset
            best_vector := (others => 0.0);
            minimum := 0.0;
        elsif (rising_edge(clk)) then  -- every clock cycle execute the algorithm
            -- if a random number is less than the input probability value then global pollination
            -- otherwise local pollination
            if (xxx < p) then
                xx := pollination_global(x_vector, levy_vector, best_vector, min, max, gamma, lamb);
            else
                xx := pollination_local(local_vector, x_vector, xxx, min, max);
            end if;
            -- using the positions found from global or local pollination, calculate the function value
            temp := ((4.0 - 2.1*(xx(1)**2) + (xx(1)**4)/3.0))*(xx(1)**2)
                              + (xx(1)*xx(2)) + (-4.0 + 4.0*(xx(2)**2))*(xx(2)**2);
                    -- check if the value is the minimum
                    -- if less change the minimum and the best positoins accordingly
                    if (temp <= minimum) then
                        minimum := temp;
                        best_vector(1) := xx(1);
                        best_vector(2) := xx(2);
                    end if;
        -- output the signal of best positions and the minimum
        best_x(1) <= best_vector(1);
        best_x(2) <= best_vector(2);
        fx <= minimum;
        end if;
    end process;
end Behavioral;
```

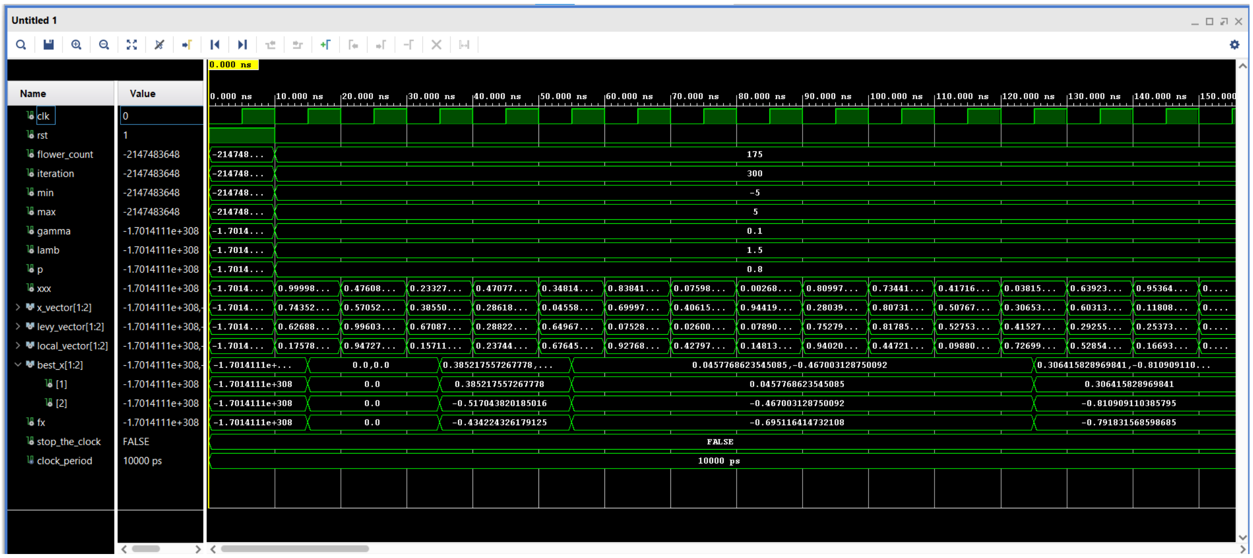
Figure 13: Test Bench Code

```
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;
use ieee.math_real.all;
use work.flower_pollination_package.all;

entity final2_tb is
end;

architecture bench of final2_tb is

  component final2
  Port ( clk, rst : in std_logic;    -- clock and reset signal
        flower_count: in integer;    -- number of flowers
        iteration : in integer;      -- number of iterations
        min, max: in integer;        -- minimum and maximum boundary
        gamma, lamb: in real;        -- gamma and lmbda value
        p: in real;                  -- probability value
        xxx : in real;               -- random real value
         -- vector with random real value for functions and equations
        x_vector,levy_vector, local_vector : in real_vector(1 to 2);
        best_x : out real_vector(1 to 2); -- vector of best values
        fx : out real );             -- best solution(minimum)
  end component;

  signal clk, rst: std_logic;
  signal flower_count: integer;
  signal iteration : integer;
  signal min, max: integer;
  signal gamma, lamb: real;
  signal p: real;
  signal xxx : real;
  signal  x_vector,levy_vector, local_vector : real_vector(1 to 2);
  signal best_x : real_vector(1 to 2);
  signal fx: real;

  constant clock_period: time := 10 ns;
  signal stop_the_clock: boolean;

```

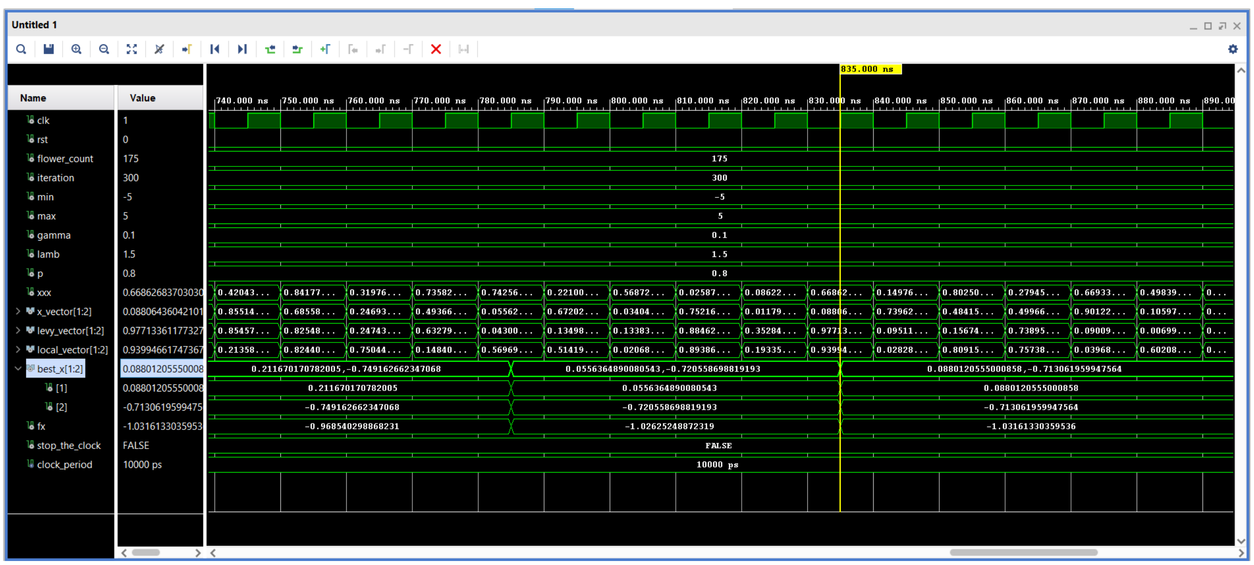
```
begin
  uut: final2 port map ( clk           => clk,
                         rst           => rst,
                         flower_count => flower_count,
                         iteration => iteration,
                         min           => min,
                         max           => max,
                         gamma         => gamma,
                         lamb          => lamb,
                         p             => p,
                         xxx           => xxx,
                         x_vector      => x_vector,
                         levy_vector   => levy_vector,
                         local_vector  => local_vector,
                         best_x  => best_x,
                         fx            => fx );

  stimulus: process
  variable seed1, seed2, seed3, seed4, seed5 : positive;
  variable xx, y, z, levy1, levy2, local1, local2 : real;
  begin
    -- reset the values
    rst <= '1';
        wait for 10 ns;
    rst <= '0';
    -- initialization for each inputs
    flower_count  <= 175;
    iteration <= 300;
    min           <= -5;
    max           <= 5;
    gamma         <= 0.1;
    lamb          <= 1.5;
    p             <= 0.8;

    -- seed numbers for random number generator
    seed1 := 1;
    seed2 := 2;
    seed3 := 3;
    seed4 := 4;
    seed5 := 5;

    for n in 1 to 300 loop
        -- using the seed values to generate random numbers each iterations
        uniform(seed1, seed2, xx);
        uniform(seed1, seed3, y);
        uniform(seed2, seed3, z);
        uniform(seed1, seed4, levy1);
        uniform(seed1, seed5, levy2);
        uniform(seed2, seed4, local1);
        uniform(seed3, seed5, local2);
        -- assigning each random values to each inputs
        xxx <= xx;
        x_vector(1)  <= y-0.002;
        x_vector(2)  <= z-0.984;
        levy_vector(1)  <= levy1;
        levy_vector(2)  <= levy2;
        local_vector(1)  <= local1;
        local_vector(2)  <= local2;
            wait for 10 ns;
    end loop;

    wait;
  end process;

  clocking: process
  begin
    while not stop_the_clock loop
      clk <= '0', '1' after clock_period / 2;
      wait for clock_period;
    end loop;
    wait;
  end process;

end;
```

Simulation Waveform

New random values are generated every clock cycle then the algorithm is executed using the random values. Figure 14:



Within 835 nanoseconds the minimum value as well as the best coordinate positions are obtained. Figure 15:



Algorithm Verification

Python

The modified Python code was used to verify the results of the FPA. In the figure below, it can be seen that the Python code converged on the same point as the VHDL code did for the minimum: -1.0316. Figure 16:

```
Iteration =  283  f(x) =  -1.0316275783783373
Iteration =  284  f(x) =  -1.0316275783783373
Iteration =  285  f(x) =  -1.0316275783783373
Iteration =  286  f(x) =  -1.0316275783783373
Iteration =  287  f(x) =  -1.0316275783783373
Iteration =  288  f(x) =  -1.0316275783783373
Iteration =  289  f(x) =  -1.0316275783783373
Iteration =  290  f(x) =  -1.0316275783783373
Iteration =  291  f(x) =  -1.0316275783783373
Iteration =  292  f(x) =  -1.0316275783783373
Iteration =  293  f(x) =  -1.0316275783783373
Iteration =  294  f(x) =  -1.0316275783783373
Iteration =  295  f(x) =  -1.0316275783783373
Iteration =  296  f(x) =  -1.0316275783783373
Iteration =  297  f(x) =  -1.0316275783783373
Iteration =  298  f(x) =  -1.0316275783783373
Iteration =  299  f(x) =  -1.0316275783783373
Iteration =  300  f(x) =  -1.0316275783783373
[0.08945476939947151, -0.7128225465183462, -1.0316275783783373]
```

Matlab

Using the Matlab code for the Six-Hump Camel Back Function to validate the exact output. Figure 17:

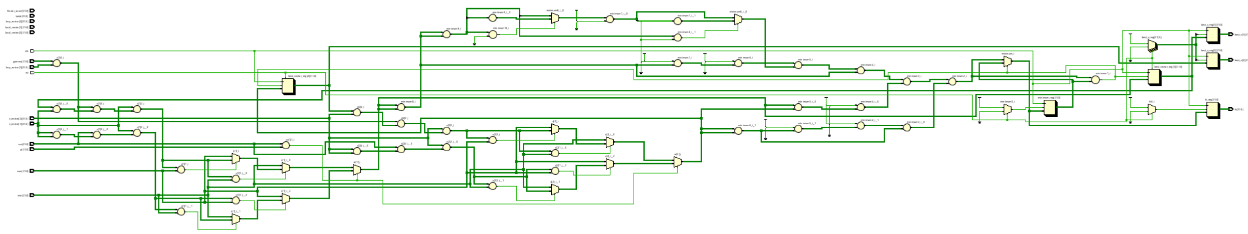
```
untitled.m
/MATLAB Drive/untitled.m
% INPUTS:
%
 xx = [0.088012, -0.71306];
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

x1 = xx(1);
x2 = xx(2);

term1 = (4-2.1*x1^2+(x1^4)/3) * x1^2;
term2 = x1*x2;
term3 = (-4+4*x2^2) * x2^2;

y = term1 + term2 + term3;

fprintf ("%d", y);
```

```
Command Window
>> untitled
-1.031613e+00
>>
```

RTL Analysis

RTL analysis required the VHDL code to have integer variables and values instead of real variables and values. In order to do that some of the functions' return types needed to be modified as well as some equations that were involved with floating numbers or division.

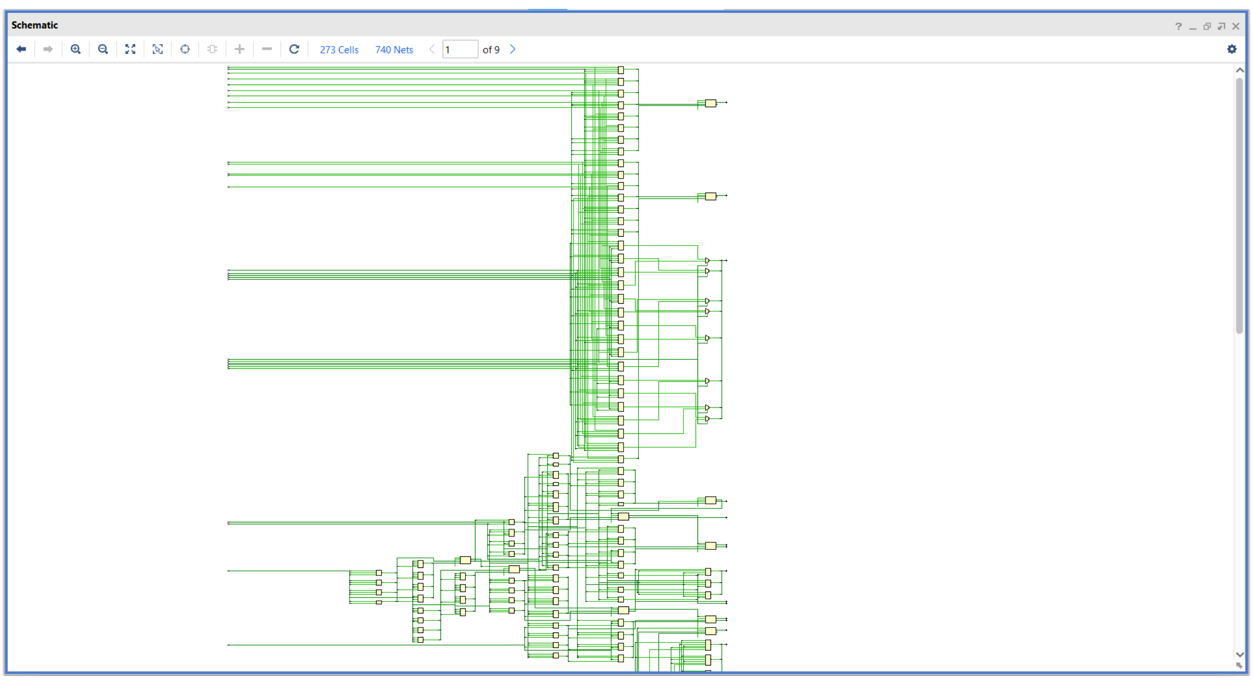
Figure 18:



Synthesized Circuit

Synthesizing required the VHDL code to have integer variables instead of real variables. In order to do that some of the functions' return types needed to be modified as well as some equations that were involved with floating numbers or division.

Figure 19:



Implemented Device

The VHDL code was not able to be implemented into the board due to the over utilization of the input and output port. This limitation may be fixed using a different model of board.

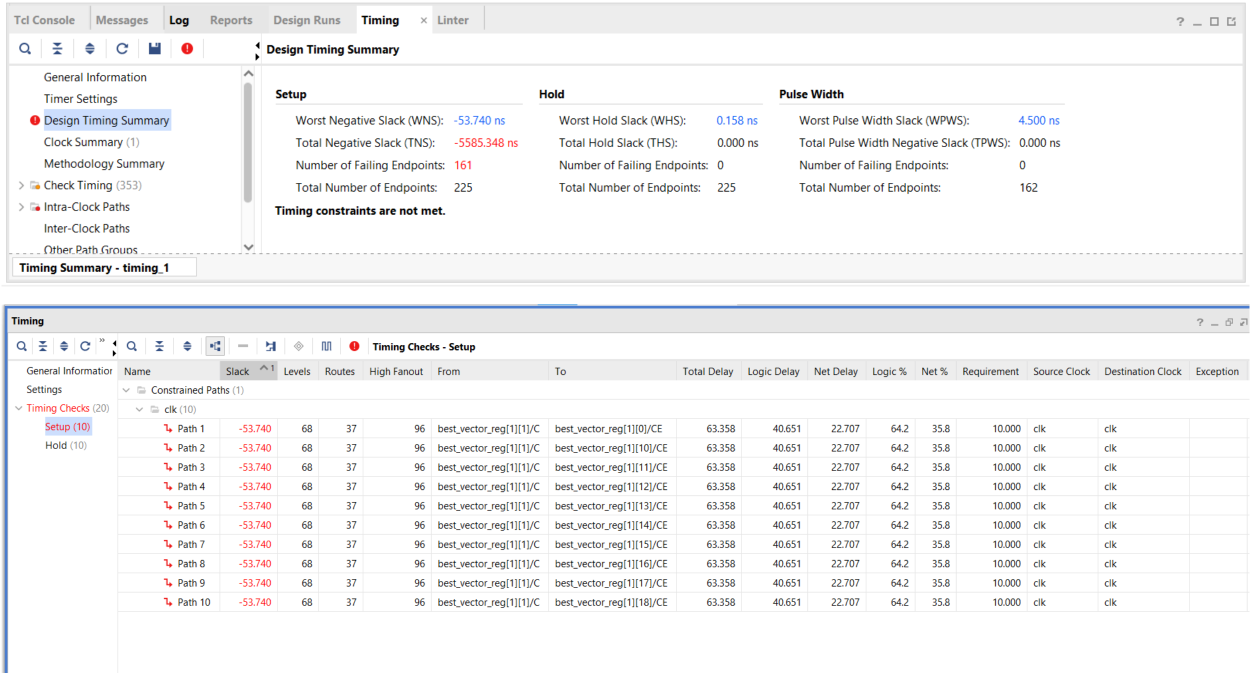
Figure 20:

Place Design (103 errors)

[Place 30-415] IO Placement failed due to overutilization. This design contains 354 I/O ports
while the target device: 7a35t package: cpg236, contains only 106 available user I/O. The target device has 106 usable I/O pins of which 0 are already occupied by user-locked I/Os.
To rectify this issue:
1. Ensure you are targeting the correct device and package. Select a larger device or different package if necessary.
2. Check the top-level ports of the design to ensure the correct number of ports are specified.
3. Consider design changes to reduce the number of I/Os necessary.

Timing Analysis

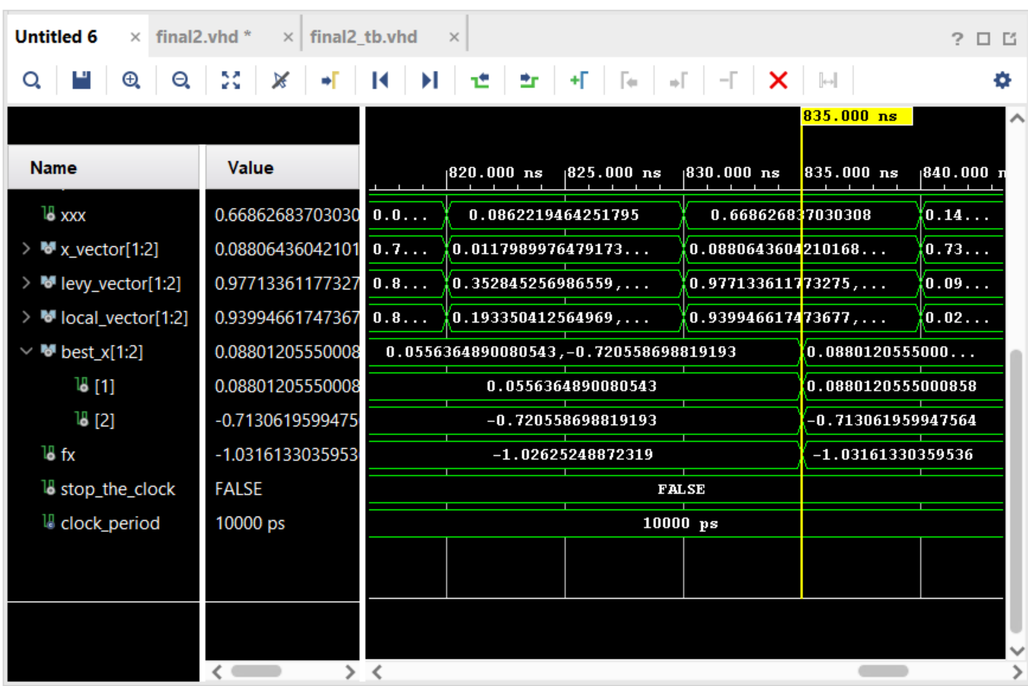
Figure 21:



Timing Information of VHDL

The VHDL code took about 835 nanoseconds and 83 iterations to find the best positions as well as the minimum values.

Figure 22:



Timing Information of Python

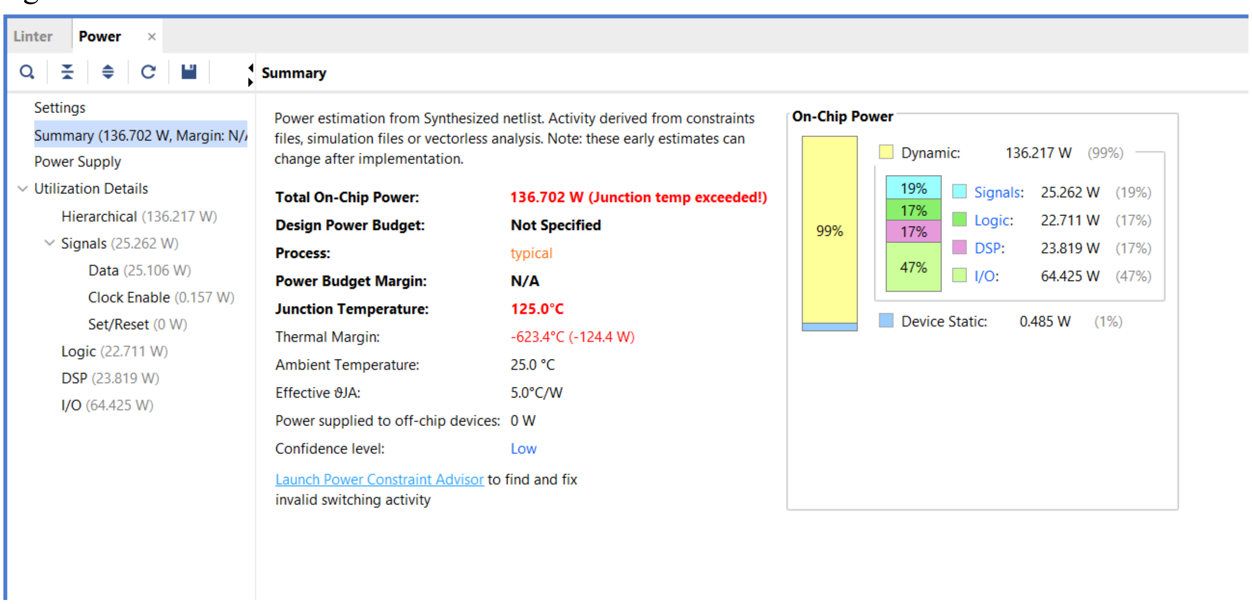
The modified Python code took about 1.05 seconds and 186 iterations to find the minimum value.

Figure 23:

```
Iteration =  181  f(x) =  -1.0315827026212248
Iteration =  182  f(x) =  -1.0315827026212248
Iteration =  183  f(x) =  -1.0315827026212248
Iteration =  184  f(x) =  -1.0315827026212248
time elapsed:  1.0451202392578125
Iteration =  185  f(x) =  -1.0315853648975715
time elapsed:  1.05312180519104

Iteration =  186  f(x) =  -1.0316095330641744
Iteration =  187  f(x) =  -1.0316095330641744
Iteration =  188  f(x) =  -1.0316095330641744
Iteration =  189  f(x) =  -1.0316095330641744
Iteration =  190  f(x) =  -1.0316095330641744
Iteration =  191  f(x) =  -1.0316095330641744
time elapsed:  1.0831239223480225
Iteration =  192  f(x) =  -1.031617853236341
Iteration =  193  f(x) =  -1.031617853236341
Iteration =  194  f(x) =  -1.031617853236341
Iteration =  195  f(x) =  -1.031617853236341
```

Power Report

Figure 24:



BASYS 3 Implementation

Finally, if the FPGA board is capable of handling the inputs and the outputs, then it can be implemented into the board as shown in the figures below.

Figure 25: Integer output of the minimum value

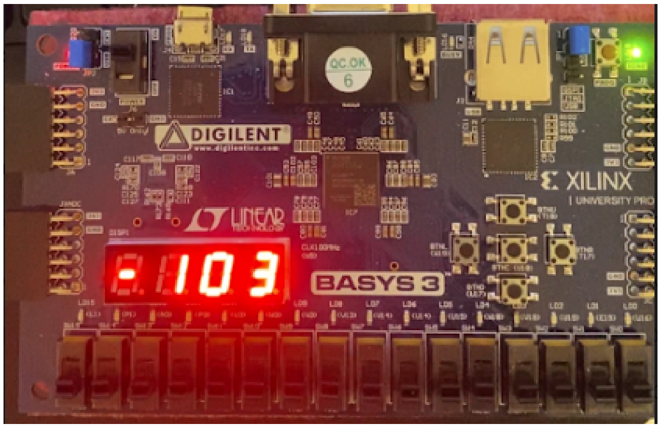
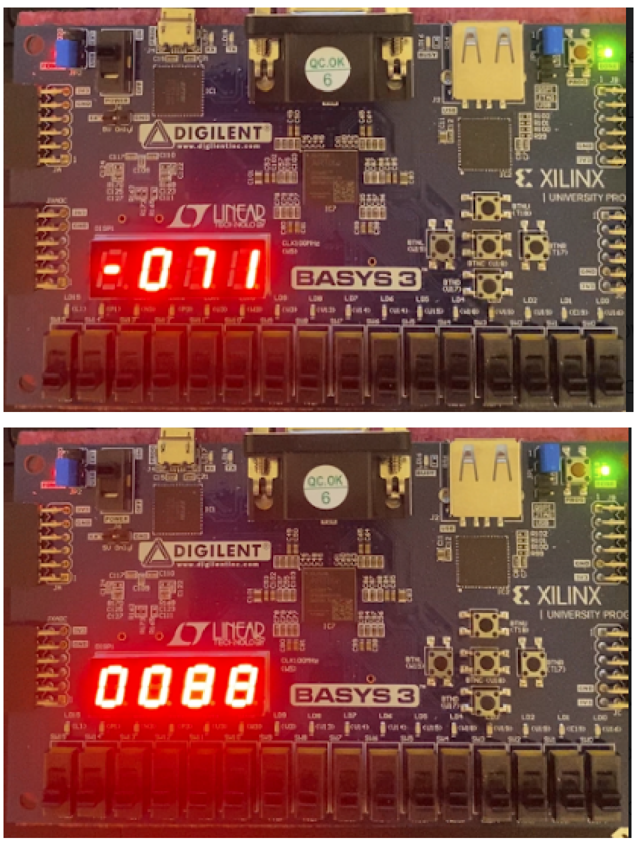


Figure 26: Integer values of two coordinate positions for the minimum output



Conclusion

The Flower Pollination Algorithm (FPA) is an effective tool for solving optimization problems, inspired by the natural pollination process. The research presented here explores the underlying principles of FPA, focusing on its advantages, applications, and limitations. We've presented several compelling, real-world applications already benefiting from this algorithm, demonstrating its wide-ranging applicability and potential. Despite its limitations, such as its computational cost and inability to adapt to changing conditions, FPA remains a valuable algorithm for tackling optimization problems. Our work in creating an FPGA-based version promises to further its reach, allowing for hardware-level implementations of this powerful algorithm. Future work should continue tuning and improving the algorithm to expand its utility in addressing more complex optimization problems.

References

[1] Mergos, P. E., & Yang, X. (2021). Flower pollination algorithm parameters tuning. *Soft Computing*, *25*(22), 14429–14447. https://doi.org/10.1007/s00500-021-06230-1

[2] Decoderz. (2020, January 27). *Flower Pollination Algorithm (FPA): A Novel Method Motivated from the Behavior of Flowers for Optimal Solution*. Transpire Online. https://transpireonline.blog/2020/01/27/flower-pollination-algorithm-fpa-a-novel-method-motivated-from-the-behavior-of-flowers-for-optimal-solution/

[3] Cui, W., & He, Y. (2018). Biological Flower Pollination Algorithm with Orthogonal Learning Strategy and Catfish Effect Mechanism for Global Optimization Problems. *Mathematical Problems in Engineering*, *2018*, 1–16. https://doi.org/10.1155/2018/6906295

[4] Subramanian, G. G., Stonier, A. A., Peter, G., & Ganji, V. (2022). Application of flower pollination algorithm for solving complex large-scale power system restoration problem using PDFF controllers. Complexity, 2022, 1–12. https://doi.org/10.1155/2022/7157524

[5] Yang, X. (2012). Flower pollination algorithm for global optimization. In Springer eBooks (pp. 240–249). https://doi.org/10.1007/978-3-642-32894-7_27

[6] Potnuru, D., Mary, K. A., & Babu, C. S. (2019). Experimental implementation of Flower

Pollination Algorithm for speed controller of a BLDC motor. Ain Shams Engineering Journal, 10(2), 287–295.https://doi.org/10.1016/j.asej.2018.07.005

[7] Suwannarongsri, S. (n.d.). Optimal solving large scale traveling transportation problems by flower pollination algorithm. https://wseas.com/journals/sac/2019/a065103-090.pdf

[8] Ali, M. M., Törn, A., & Viitanen, S. (2005). A numerical comparison of some modified controlled random search algorithms. Journal of global optimization, 31(4), 545-572.

[9] Al-Sadi, A. M., & Budayan, M. (2016). Solving the global optimization problem using Chebyshev interpolation and metaheuristic techniques. International Journal of Operations Research, 12(1), 47-59.

[10] Gandomi, A. H., Yang, X. S., Alavi, A. H., & Talatahari, S. (2013). Flower pollination algorithm for solving constrained optimization problems. Journal of Intelligent & Fuzzy Systems: Applications in Engineering and Technology, 24(1), 27-35.