REAL CODED GENETIC ALGORITHM AND SELF-ORGANIZING HIERARCHICAL PARTICLE SWARM OPTIMIZATION FOR OPTIMAL SHORT-TERM HYDROTHERMAL SCHEDULING

by

Franced Amiel G. Nidar

A special problem submitted to the

Department of Mathematics and Computer Science

College of Science

The University of the Philippines

Baguio, Baguio City

as partial fulfillment of the

requirements for the degree of

Bachelor of Science in Computer Science

May 2015

This is to certify that this Special Problem entitled **" Real Coded Genetic Algorithm and Self-Organizing Hierarchical Particle Swarm Optimization for Optimal Short-Term Hydrothermal Scheduling"**, prepared and submitted by **Franced Amiel G. Nidar** to fulfill part of the requirements for the degree of **Bachelor of Science in Computer Science**, was successfully defended and approved on May 2015.

Joel M. Addawe, Ph.D.
Special Problem Adviser

The Department of Mathematics and Computer Science endorses the acceptance of this Special Problem as partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science .

Jerico B. Bacani, Ph.D.
Chair
Department of Mathematics and
Computer Science

Table of Contents

Acknowledgments vi

Abstract viii

List of Tables xi

List of Figures xii

Chapter 1. Introduction 1
1.1 Background of the Study 1
1.2 Statement of the Problem 2
1.3 Objective of the Study 3
1.3.1 General Objective 3
1.3.2 Specific Objective 3
1.4 Significance of the Study 3
1.5 Scope and Limitation 4

Chapter 2. Review of Related Literature 5
2.1 Thermal Plant 5
2.1.1 British Thermal Unit (Btu) 6
2.1.2 Boiler 7
2.1.3 Turbine 7
2.1.4 Condenser 8
2.1.5 Electric generator 9
2.1.6 Cooling tower 9
2.1.7 Mathematical Model 10
2.2 Hydro Plant 20
2.2.1 Reservoir 21
2.2.2 Water Intake/Control Gates 21
2.2.3 Penstock 21
2.2.4 Water Turbines 22
2.2.5 Electric Generator 22
2.2.6 Mathematical Model 23
2.3 Short-term Hydrothermal Scheduling(SHS) 29
2.3.1 Data Representation and Decision Variables 30
2.3.2 Initialization of Population 30
2.4 Genetic Algorithm(GA) 31
2.4.1 Operators of GA 32

2.5 Real Coded Genetic Algorithm(RCGA) 35
2.5.1 Operators of RCGA 35
2.5.2 Next Generation 40
2.5.3 Stopping Criterion 40
2.6 Artificial Fish Swarm Algorithm (AFSA) 40
2.6.1 Components of AFSA 41
2.7 Particle Swarm Optimization (PSO) 42
2.7.1 Velocity 42
2.7.2 Cognitive Factor 43
2.7.3 Social Factor 43
2.7.4 Time Varying Inertia Weight (TVIW) 44
2.7.5 Time Varying Acceleration Coeffcients (TVAC) 44
2.8 Self-Organizing Hierarchical Particle Swarm Optimization with Time Varying Acceleration Coefficients 44

Chapter 3. Methodology 46
3.1 Parameter Constants 46
3.1.1 Test System 1 46
3.1.2 Test System 2 48
3.2 Data Representation 48
3.3 Initialization of Population 49
3.4 Evaluation 50
3.5 Constraints Handling 51
3.5.1 Hydro Plants 51
3.5.2 Thermal Plants 54
3.6 Real-Coded Genetic Algorithm 55
3.6.1 Selection 56
3.6.2 Crossover 57
3.6.3 Mutation 59
3.7 Artificial Fish Swarm Algorithm 60
3.7.1 Swarm Behavior 60
3.7.2 Follow Behavior 63
3.7.3 Prey behavior 64
3.8 Particle Swarm Optimization with time-varying inertia weight 65
3.8.1 Initialization of Velocity 66
3.8.2 Updating the Velocity 67
3.8.3 Updating the population 69
3.9 Self-Organizing Hierarchical Particle Swarm Optimization with time-varying acceleration coefficients 70
3.9.1 Initialization of Velocity 70
3.9.2 Updating the Velocity 72
3.9.3 Updating the population 73

3.10 Plan of Operations 74

Chapter 4. Results and Discussion 76
4.1 Methods 76
4.1.1 RCGA 76
4.1.2 RCGA-AFSA 76
4.1.3 RCGA-PSO 76
4.1.4 RCGA-SOHPSO 76
4.2 Results 77
4.2.1 Reservoir Volume Violations 79

Chapter 5. Conclusion and Recommendation 80
5.0.2 Recommendation 80

List of References 81

Appendix A.Table for Test System 1, Case 1 86

Appendix B.Table for Test System 1, Case 2 92

Appendix C.Table for Test System 2, Case 1 98

Appendix D.Table for Test System 2, Case 2 104

Appendix E.Reservoir Volume Violations 109
E.1 RCGA 110
E.2 RCGA-AFSA 110
E.3 RCGA-PSO 111
E.4 RCGA-SOHPSO 111

Appendix F.Power Generation Violations 112

Appendix GSource Code for Test System 1, all cases 113

Appendix HSource Code for Test System 2, all cases 139

Acknowledgments

This study would not have been possible without the people around me.

I express my sincere thanks to my parents for supporting me throughout this journey.

I would also like to express my gratitude to our adviser, Dr. Joel Addawe, without whom I would not be able to accomplish this paper.

I express my thanks to the Department of Mathematics and Computer Science especially Sir Reymart S. Lagunero for helping in a certain part of this paper.

To my friends for listening to my self-reflection and answering some inquiries regarding this paper, thank you.

And last but not the least, I would like to thank the Lord for the ability and opportunity to contribute.

Abstract

Real Coded Genetic Algorithm and Self-Organizing Hierarchical Particle Swarm Optimization for Optimal Short-Term Hydrothermal Scheduling

Franced Amiel G. Nidar
University of the Philippines, 2015

Adviser:
Joel M. Addawe, Ph.D.

Short-term hydrothermal scheduling(SHS) [19] has been one of the main issues in operating systems of hydro and thermal plants because of their complexity and nonlinearity. Solving the SHS problem using heuristic methods can produce a more effcicient set of solution to minimize the consumption of fuel and thus lowering the cost of providing electricity to the people. SHS problem includes many equality and inequality constraints, mainly hydraulic constraints, reservoir water volume, power balance constraints, thermal generation limits, and water discharge rate limits of hydro plants. Having a complex nature makes the SHS problem difficult to solve using only conventional methods.

In this paper we present two exisiting algorithms and two proposed algorithms. The two existing algorithms are real coded genetic algorithm (RCGA) and a hybrid of real coded genetic algorithm and artificial fish swarm algorithm (RCGA-AFSA). The two proposed methods are a hybrid of real coded genetic algorithm and particle swarm optimization (RCGA-PSO) and a hybrid of real coded genetic algorithm and self-organizing hierarchical particle swarm optimization (RCGA-SOHPSO), respectively, in the optimisation of optimal solution set of thermal and hydro plants. We present two short-term hydrothermal scheduling problem test system in which the methods are used. The first test system consists of 4 hydro plants and 1 thermal plant. The second test system consists of 4 hydro plants equal to the hydro plants of the first test system and 3 thermal plant. The proposed hybrid algorithm that use self-organizing hierarchical particle swarm optimization yields better results in test system 1 case 1 with a fitness value of $892154 and in test system 1 case 2 with a fitness value of $899068 compared to the existing

algorithms and RCGA-PSO. The results also concluded that swarming algorithms are good for less complex systems and their effectivity lessens as the system becomes more complex and more variables are considered.

List of Tables

2.1 Data gathered from Moss Landing Unit 7. A table of incremented intervals of 15 in power output (MW) and their corresponding input(input-output curve), incremental heat rates, and average heat rates. 13

2.2 INPUT-OUTPUT CURVE FOR MOSS LANDING UNIT 7 IN INCREMENTS of 0.1 15

3.1 Constants of thermal plants for test system 1 47

3.2 Water reservoir inflow for all test system 47

3.3 Example parameters 74

A.1 Test System 1, Case1, RCGA: Average Best Chromosome,Average best fitness = 908215, Average worst fitness = 940105, CPU time = 608.35 secs 87

A.2 Average Best P_h 87

A.3 Test System 1, Case1, RCGA-AFSA: Average Best Chromosome,Average best fitness = 904338, Average worst fitness = 927770, CPU time = 967.83 secs 88

A.4 Average Best P_h 88

A.5 Test System 1, Case1, RCGA-PSO: Average Best Chromosome,Average best fitness = 914646, Average worst fitness = 944457, CPU time =984.95 secs 89

A.6 Average Best P_h 89

A.7 Test System 1, Case1, RCGA-SOHPSO: Average Best Chromosome,Average best fitness = 892154, Average worst fitness = 936900, CPU time =964.5 secs 90

A.8 Average Best P_h 90

B.1 Test System 1, Case2, RCGA: Average Best Chromosome,Average best fitness = 917760, Average worst fitness = 951885, CPU time = 607.21 secs 93

B.2 Average Best P_h 93

B.3 Test System 1, Case2, RCGA-AFSA: Average Best Chromosome,Average best fitness = 914890, Average worst fitness = 940596, CPU time =959.84 secs 94

B.4 Average Best P_h 94

B.5 Test System 1, Case2, RCGA-PSO: Average Best Chromosome,Average best fitness = 925355, Average worst fitness = 955162, CPU time =982.41 secs 95
B.6 Average Best P_h 95
B.7 Test System 1, Case2, RCGA-SOHPSO: Average Best Chromosome,Average best fitness = 899068, Average worst fitness = 946753, CPU time =963.69 secs 96
B.8 Average Best P_h 96

C.1 Test System 2, Case1, RCGA: Average Best Chromosome,Average best fitness = 44926.8, Average worst fitness = 49811.5, CPU time = 946.33 99
C.2 Average Best P_h 99
C.3 Test System 2, Case 1, RCGA-AFSA: Average Best Chromosome, Average best fitness = 45842.2, Average worst fitness =49590.3 , CPU time = 1289.15 secs 100
C.4 Average Best P_h 100
C.5 Test System 2, Case 1, RCGA-PSO: Average Best Chromosome, Average best fitness = 46442.9, Average worst fitness =76115.4, CPU time = 1386.52 secs 101
C.6 Average Best P_h 101
C.7 Test System 2, Case 1, RCGA-SOHPSO: Average Best Chromosome, Average best fitness = 46320.8, Average worst fitness =54066.6, CPU time = 1347.1 secs 102
C.8 Average Best P_h 102

D.1 Test System 2, Case 2, RCGA: Average Best Chromosome, fitness = 45818.6, Average Worst Fitness = 49850.9, CPU time = 1424.73 secs 105
D.2 Average Best P_h 105
D.3 Test System 2, Case 2, RCGA-AFSA: Average Best Chromosome, Average Best Fitness = 44490.7, Average Worst Fitness = 49972.7, CPU time = 1504.63 secs 106
D.4 Average Best P_h 106
D.5 Test System 2, Case 2, RCGA-PSO: Average Best Chromosome, fitness = 46455.1, CPU time = 1424.73 secs 107
D.6 Average Best P_h 107
D.7 Test System 2, Case 2, RCGA-SOHPSO: Average Best Chromosome, Average Best Fitness = 46333.8, Average Worst Fitness = 51146.1, CPU time = 1424.73 secs 108
D.8 Average Best P_h 108

List of Figures

2.1 A schematic diagram of a thermal plant.[27] 6
2.2 A SCADA diagram of a boiler obtained from Schneider Electric IGSS . . 7
2.3 A steam turbine used in thermal plants. The steam enters the turbine following the direction of the arrow and flows through a series of blades. The steam is then expelled from the turbine via the exit.[1] 7
2.4 A simple alternator that uses magnetic induction to produce electricity. N and S are magnets and denote north and south, respectively. B is the magnetic field exerted by the two magnets. I is the current produced when the change in B exerts a flow of electrons. [2] 9
2.5 1 thermal plant for time interval $t = 0$ to T 10
2.6 N_s thermal plant for time interval $t = 0$ to T 11
2.7 INPUT-OUTPUT CURVE for MOSS LANDING UNIT 7 from Table 2.1 gathered from report [26] 14
2.8 INPUT-OUTPUT CURVE of MOSS LANDING 7 [26] 15
2.9 Fuel cost coefficients of Moss Landing 7 from Table 2.1 produced by MATLAB 16
2.10 Fuel cost coefficients of Moss Landing 7 from Table 2.2 produced by MATLAB 16
2.11 valve-point effects as described in [33] 17
2.12 An overview of the major parts of a hydro plant namely the water reservoir, water intake, penstock, turbine, powerhouse, and electric generator [3] . . 21
2.13 Different types of water turbines and their general efficiency using the input-output curve [33] 22
2.14 Different types of water turbines. From left to right: Pelton Wheel, two types of Francis turbine, and Kaplan turbine [4] 22
2.15 A typical elevation-storage curve which takes the form of an inverted second order polynomial function [33] 23
2.16 Trapezoidal reservoir configuration [29] 24

2.17 Hydro Plant Schematics. A representation of reservoir with their own powerhouse, inflow rate $I_h(j,t)$, spillage rate $S_h(j,t)$ and water discharge rate $Q_h(j,t)$. 27
2.18 Generalized flow of the GA process 32

3.1 Flowchart of real-coded genetic algorithm 56
3.2 Flowchart of artificial fish swarm algorithm 60
3.3 Flowchart of particle swarm optimization 65

E.1 Hourly reservoir storage volumes obtained by RCGA 110
E.2 Hourly reservoir storage volumes obtained by RCGA-AFSA 110
E.3 Hourly reservoir storage volumes obtained by RCGA-PSO 111
E.4 Hourly reservoir storage volumes obtained by RCGA-SOHPSO 111

Chapter 1

Introduction

Electricity is plays a major role to all developing communities and nations due to the rising dependence of economies and people to electricity-driven industries. Electric power is produced by electric generators found in thermal, hydro, geothermal and wind plants. The production of electricity is crucial as well as the planning of the production of electric power. In the process of the production of electricity in many sources like thermal and hydro plant for example, planning the output of these power plants and satisfying their corresponding system constraints is vital to minimize the cost. The problem in particular with two of the most used type of power plants, namely thermal and hydro plants, is called hydrothermal scheduling. Planning the economic dispatch of ahydrothermal system that ranges from hour to hour for a day is called short-term hydrothermal scheduling (SHS).

1.1 Background of the Study

A hydrothermal plant system consists of several individual hydro and thermal plants operating simultaneously with each other to produce electricity and meet a power demand. Meeting the power demand while optimally discharging the water of the reservoir in a hydro plant and minimally using fuels in thermal plants is an issue in operating the hydrothermal plant system. SHS optimization has been thoroughly studied because of its beneficial applications to curve down the use of dependable non-renewable fuels and promote the use of renewable energy. Since renewable energy(e.g. hydro, solar, wind) does not satisfy the security for supplying power due to its fluctuating sources of energy, the usual setup of a power distribution system employs the use of non-renewable energy alongside the renewable energy and thus the importance of optimizing SHS.

Short-term hydrothermal scheduling is important to operating power distribution systems consisting of hydro and thermal plants. SHS problem includes equality and inequality constraints specifically hydraulic constraints, reservoir water volume, power balance constraints, thermal generation limits, and water discharge rate limits of hydro plants.

The optimal settings of the hydrothermal plant system include the maximized water discharge rate while conserving the water in the reservoir and the minimized power generation which leads to less fuel consumption.

The first stage of developing an efficient and effective algorithm for the SHS problem is defining and conceptualizing a mathematical model for the operation of thermal and hydro plants.

A more detailed background is given in Section 2 Review of Related Literature.

1.2 Statement of the Problem

Determining an optimal setting for the thermal and hydro plant to minimize the fuel cost of the thermal plants is the main objective of the short-term hydrothermal scheduling using combinations of hybrid heuristic methods namely, real coded genetic algorithm (RCGA), real coded genetic algorithm and artificial fish swarm algorithm (RCGA-AFSA), real coded genetic algorithm and particle swarm optimization (RCGA-PSO), and real coded genetic algorithm and self organizing hierarchical particle swarm optimization (RCGA-SOHPSO).

Existing mathematical models for thermal and hydro plant, and a constraints handling method are presented in Section 2.

1.3 Objective of the Study

1.3.1 General Objective

The general objective of the study is to contribute to the growing demand in power. The study aims to minimize the fuel consumption of thermal plants and thus contributing both to economic growth and environmental concerns of our country.

1.3.2 Specific Objective

The specific objective of the study is to minimize the fuel cost of thermal plants while maximizing the power produced by the hydro plants. Hydro plant equality and inequality constraints are considered. A series of hydraulic and electric system constraints for the thermal plant and hydro plant are also considrered.

1.4 Significance of the Study

A report from the Philippines' Department of Energy states that 60% of Philippines' energy comes from non-renewable sources as of 2011 in the category of primary energy supply [40]. Fuels for thermal plants include but are not limited to oil, coal, and natural gas. Determining the minimal fuel consumption in producing energy will have beneficial effects for the country. The cost in providing energy to the population will lessen since the fuel consumption is minimized. Reduction of carbon emission by the burning of fuels will also bring environmental benefit and lessen the effects of global warming. Since DoE plans on making hydropower the main source of electricity by 2030 and considering the fact that thermal plants are the most dependable source of energy, striking a balance between the hydro and thermal plants should be made.

1.5 Scope and Limitation

The paper aims to produce a more suitable method for solving the SHS optimization problem through sets of optimal settings of thermal power generation and water discharge of hydro plants given a set of power demand,inequality constraints and equality constraints. The paper includes a brief discussion of mathematical models of thermal and hydro plants and does not aim to produce new optimal parameters and new mathematical models for hydro and thermal plants. It does however compare the methods used, namely RCGA, RCGA-AFSA, RCGA-PSO, and RCGA-SOHPSO. The study also show how the three swarming algorithms, PSO, AFSA, and SOHPSO behave in a complex multi-objective optimization problem.

The algorithms were written in C++ 4.8.1 on computer with Intel i3 3rd Generation, with 4GB of RAM running on Windows 7 Ultimate. The algorithms were executed on computers with Intel i5 3.3GHz, with 4GB of RAM running on Linux.

Chapter 2

Review of Related Literature

The Review of Related Literature (RRL) focuses on the different terminologies regarding hydro and thermal plants. The mathematical models of the hydrothermal plant system are defined in their respective type of power plant. The limits and constraints of variables are also defined in this section. A new constraints handling method for both hydro and thermal plants presented in [19] is also briefly discussed.

The RRL expounds the structure and components of the heuristic methods used in solving the SHS problem, namely real coded genetic algorithm (RCGA), artificial fish swarm algorithm (AFSA), particle swarm optimization (PSO), and self-organizing hierarchical particle swarm optimization (SOHPSO).

2.1 Thermal Plant

A thermal plant is a collection of machines and components that converts fuels to generate electric power. These fuel are often in the form of coal, oil, or gas and the the fuels are usually measured in MBtu/hr or MJ/hr.

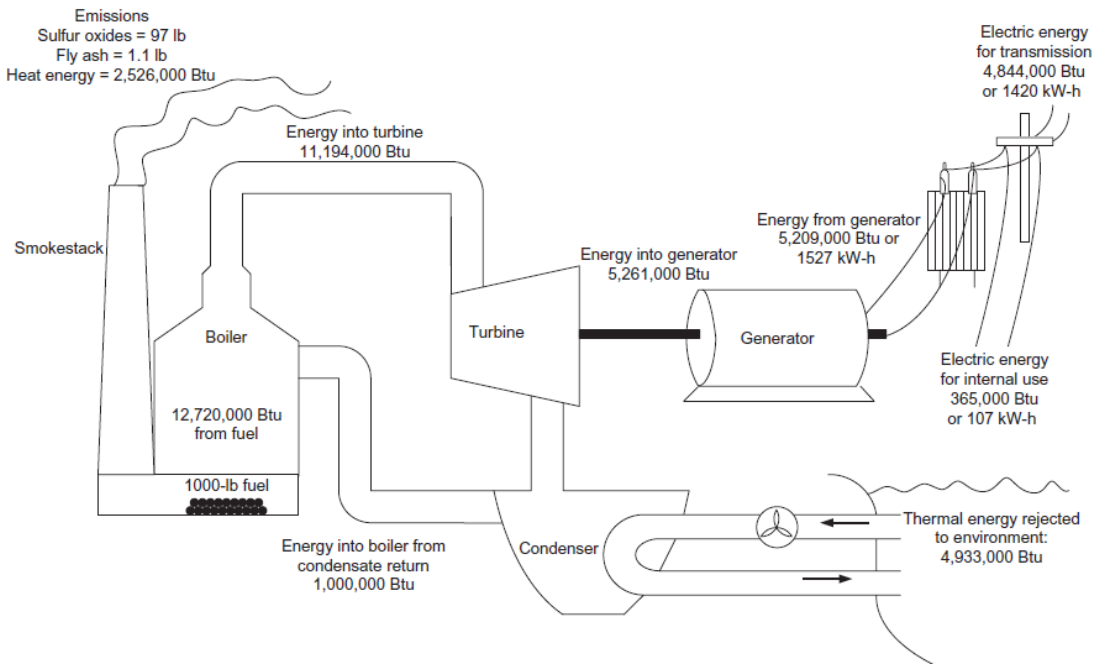


Figure 2.1: A schematic diagram of a thermal plant.[27]

The major parts of a thermal plant as can be seen in Figure 2.1 are the boiler, turbine, electric generator, condenser, and cooling tower.

2.1.1 British Thermal Unit (Btu)

Btu is a standard measurement of input in types of power plants where the burning of fuel to produce electricity is used. Btu is the amount of heat required to heat a pint of water by one degree Fahrenheit. 1 Btu is equal to 1055 joules.

2.1.2 Boiler

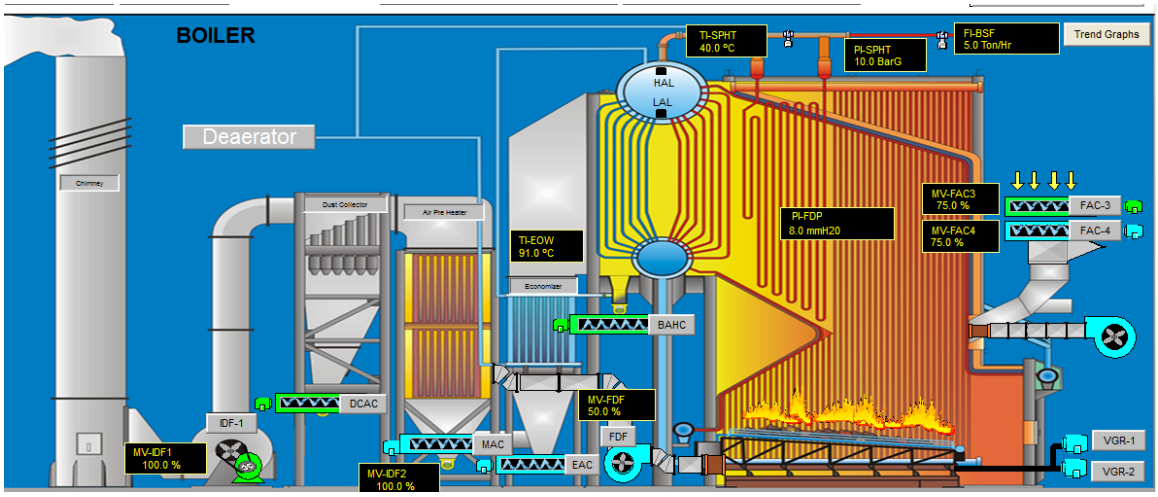


Figure 2.2: A SCADA diagram of a boiler obtained from Schneider Electric IGSS

The boiler is responsible for creating high pressure steam used by the thermal plant. When water is heated through the burning of fuels, the water expands and converted to high pressure steam. The high pressure steam is responsible for turning the turbine to produce mechanical energy that turns the electric generator to produce electric power.

2.1.3 Turbine

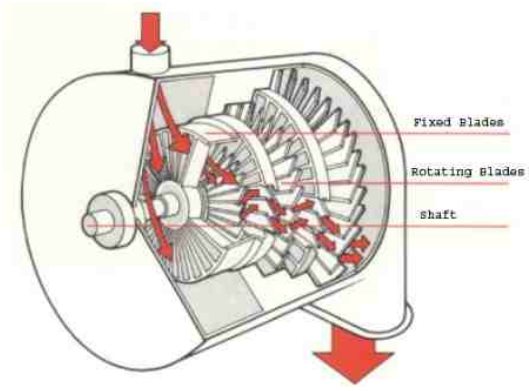


Figure 2.3: A steam turbine used in thermal plants. The steam enters the turbine following the direction of the arrow and flows through a series of blades. The steam is then expelled from the turbine via the exit.[1]

The turbine is a device that converts the high pressure steam from the boiler into mechanical energy through a series of blades. As seen on Figure 2.3, the flow of steam depicted by the arrows turn the blades and thus producing mechanical energy. The high pressure steam enters the contraption with blades inside and once pressure builds up, the steam will find an exit to relieve the pressure. The exit in the other side of the turbine will serve as the pressure reliever as the steam passes through the blades. Often the steam is reheated and is passed through another turbine to extract more energy from the steam.

2.1.4 Condenser

After the high pressure steam go through the boiler and turbine, it goes directly to the condenser. The condenser condenses steam back to its liquid state. The process of condensing the steam back to liquid is done to obtain maximum efficiency and conserve water. The water that comes from the condenser is cooled and fed back again to the boiler. The process of condensing steam into water is analogous when steam cool down and some water condenses at the sides of the container.

2.1.5 Electric generator

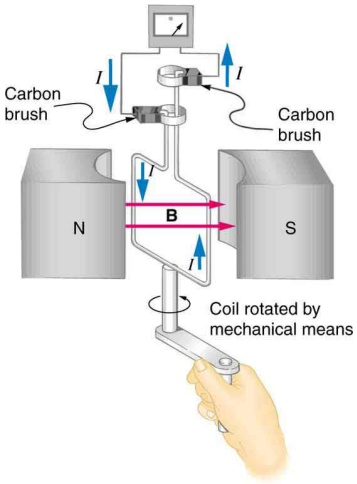


Figure 2.4: A simple alternator that uses magnetic induction to produce electricity. *N* and *S* are magnets and denote north and south, respectively. B is the magnetic field exerted by the two magnets. I is the current produced when the change in B exerts a flow of electrons. [2]

An electric generator converts mechanical energy from the turbine to electrical energy through magnetic induction. Figure 2.4 shows how magnetic induction produces electricity. Magnetic induction is made if there is a change of magnetic field called magnetic flux, there will be an induced electromotive force and thus a flow of electrons known as current is produced. The phenomena of flowing electrons is called electricity. The carbon brush then conducts the current produced from the moving coil to the stationary part of the electric generator and is delivered to households. An electric generator falls into two main categories, namely alternator and dynamo. Alternators generate alternating current while dynamos and commutators produce direct current.

2.1.6 Cooling tower

The water that came from the condenser is initially near the boiling point of water. The hot water is passed on cooling towers to cool down. There are two ways to cool the water. The first way is through the evaporation process which employs air in direct

contact with the water to transfer the heat. In evaporative setup, an area is open where the water is being cooled. The heat from the water is transferred to the surrounding air through evaporation. The second is through non evaporative cooling which uses indirect contact with the water but transfers heat nonetheless. The non-evaporative setup is analogous to having a radiator wherein the water travels through the coils. The coil is in contact with the surrounding air. The heat is transfered from the water to the metal of the coil then to the surrounding air.

2.1.7 Mathematical Model

The mathematical model of the thermal plants is based on the book of El-Hawary and Christensen[33] which tackles different configuration and types of hydrothermal systems.

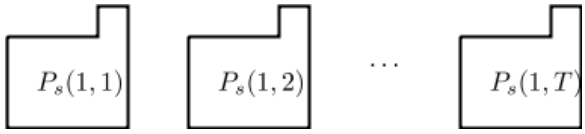


Figure 2.5: 1 thermal plant for time interval $t = 0$ to T

The objective function for the SHS problem is the minimization of the fuel consumption of the thermal plant while considering the power demand and its constraints. Fuel consumption can be derived from energy expended by the thermal plants. Assigning first a variable for power generation to derive a fuel cost function, power generation is a set of data of 1 to i^{th} thermal plant divided evenly by hourly time interval ranging from 1^{st} to 24^{th} hour.
Power generation of thermal plants is denoted by:

$$P_s(i,t)$$

The more fuel a thermal plant burns, the more energy it produces and thus, fuel cost is a function of power generated by a thermal plant which can be denoted in the same way in Figure 2.5 as :

$$F_1 = f_1(P_s(1,1)) + f_1(P_s(1,2)) + \ldots + f_1(P_s(1,T)) \quad (1)$$

where f_t is the fuel cost function, t is the time interval from 1 to T, and F_1 is the total fuel cost function of the 1^{st} hydro plant.

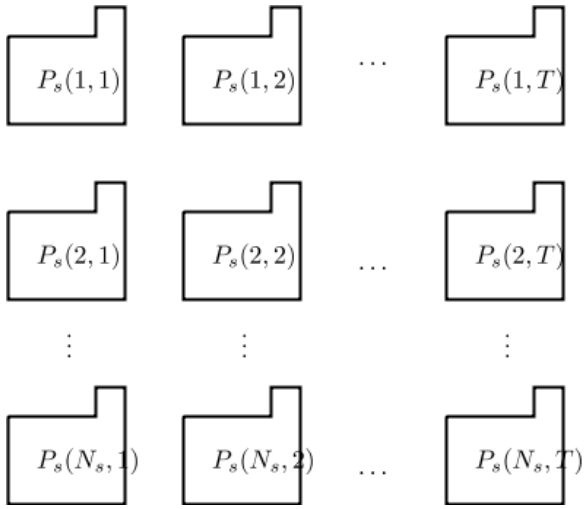


Figure 2.6: N_s thermal plant for time interval $t = 0$ to T

Since the model should accomodate many thermal plants, the total cost function can be derived as:

$$F_{totalcost} = F_1 + F_2 + \ldots + F_{N_s} \tag{2.1}$$

where N_s is the number of total thermal plants in a system and $F_{totalcost}$ is the total fuel cost of all thermal plants in all time interval.
Thus, if we consider equations (1) and (2.1), $F_{totalcost}$ is equal to

$$F_{totalcost} = \sum_{t=1}^{T} \sum_{i=1}^{N_s} f_i(P_s(i,t)) \tag{2.2}$$

In defining f_i, other factors are considered since the conversion of fuel to energy in a thermal plant is a process wherein many variables affect the conversion. The fuel cost function can only be achieved through approximation due to its complex nature. A technical report of Joel B. Klein[26] gives us actual data to work with a suitable form of fuel cost function. There are two main variables in dealing with the cost function. The first variable is the input, which can vary from the fuel being fed to the thermal plant,

which is in units of heat (Btu), or units of energy(Joule). The input variable may vary and is interchangeable with other units as long as it remains as an input in a thermal system. For uniformity, we will use the unit of heat which is also the actual data recorded from [26]. The second variable is the output, which is the electricity measured in watts. The data provided by [26] is from Moss Landing Unit 7, a natural gas thermal plant from California. The Moss Landing 7 is one of the most efficient thermal plant and will suffice in approximating the fuel cost function. The data gathered are shown in the Table 2.1.

	cap (MW)	input-output curve (1000 Btu/hr)	incremental heat rates (btu/kWh)	average heat rates (Btu/kWh)
Block 1	50	997,950	6,847	19,959
	65	1,100,631	6,929	16,933
	80	1,205,167	7,009	15,065
	95	1,310,893	7,087	13,799
	110	1,417,782	7,164	12,889
	125	1,525,808	7,239	12,206
	140	1,634,944	7,312	11,678
	155	1,745,164	7,384	11,259
	170	1,856,442	7,453	10,920
Block 2	185	1,968,751	7,521	10,631
	200	2,082,065	7,587	10,410
	215	2,196,358	7,652	10,216
	230	2,311,603	7,714	10,050
	245	2,427,775	7,775	9,909
	260	2,544,846	7,834	9,788
	275	2,662,791	7,892	9,683
	290	2,781,583	7,947	9,592
	305	2,901,195	8,001	9,512
	320	3,021,603	8,053	9,443
	335	3,142,778	8,103	9,381
	350	3,264,695	8,152	9,328
Block 3	370	3,428,360	8,214	9,268
	385	3,551,905	8,258	9,226
	400	3,676,105	8,301	9,190
	415	3,800,932	8,342	9,159
	430	3,926,361	8,381	9,131
	445	4,052,365	8,419	9,106
	460	4,178,918	8,455	9,085
	475	4,305,993	8,489	9,065
	490	4,433,565	8,521	9,048
	505	4,561,606	8,551	9,033
	520	4,690,091	8,580	9,019
	535	4,818,992	8,607	9,007
	550	4,948,285	8,632	8,997
	565	5,077,942	8,655	8,988
	580	5,207,937	8,677	8,979
Block 4	591	5,303,467	8,692	8,962
	606	5,433,986	8,710	8,967
	621	5,564,771	8,727	8,961
	636	5,695,797	8,742	8,956
	651	5,827,035	8,756	8,951
	666	5,958,461	8,767	8,947
	681	6,090,048	8,777	8,943
	696	6,221,770	8,785	8,939
	711	6,353,600	8,792	8,936
	726	6,485,511	8,796	8,933
Block 5	739	6,599,881	8,799	8,971

Table 2.1: Data gathered from Moss Landing Unit 7. A table of incremented intervals of 15 in power output (MW) and their corresponding input(input-output curve), incremental heat rates, and average heat rates.

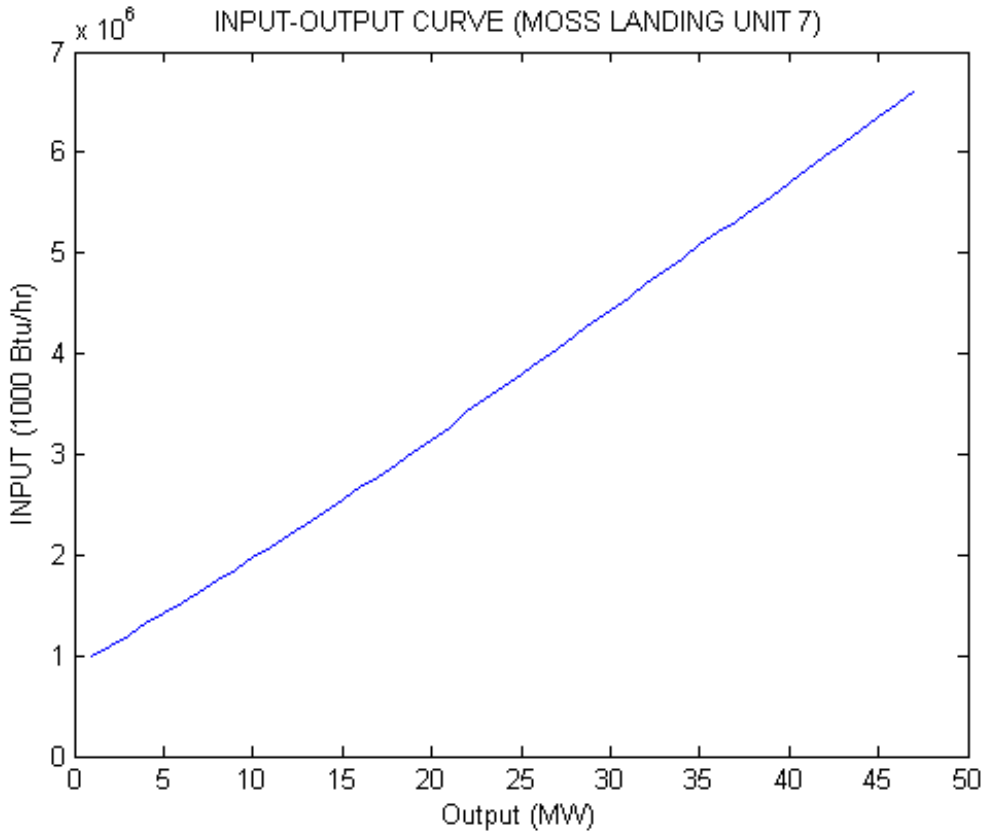


Figure 2.7: INPUT-OUTPUT CURVE for MOSS LANDING UNIT 7 from Table 2.1 gathered from report [26]

The data in Table 2.1 is plotted and the resulting graph results in the input-output curve in Figure 2.1. The relationship between the input and output shows an almost linear relation since the efficiency of the Moss Landing Unit 7 is high. Having a linear relation between the input and output means that the fuel is efficiently converted to electric energy. Another set of data which is in increments of 0.1 MW are gathered from the same unit to understand the behavior of the input-output curve.

cap (MW)	input-output curve (1000 Btu/hr)	incremental heat rates (btu/kWh)	average heat rates (Btu/kWh)
1.0	20,000	3,000	20,000
1.1	20,310	3,200	18,464
1.2	20,640	3,400	17,200
1.3	20,990	3,600	16,146
1.4	21,360	3,800	15,257
1.5	21,750	4,000	14,500
1.6	22,160	4,200	13,850
1.7	22,590	4,400	13,288
1.8	23,040	4,600	12,800
1.9	23,510	4,800	12,374
2.0	24,000	5,000	12,000
2.1	24,510	5,200	11,671
2.2	25,040	5,400	11,382
2.3	25,590	5,600	11,126
2.4	26,160	5,800	10,900
2.5	26,750	6,000	10,700
2.6	27,360	6,200	10,523
2.7	27,990	6,400	10,367
2.8	28,640	6,600	10,229
2.9	29,310	6,800	10,107
3.0	30,000	7,000	10,000

Table 2.2: INPUT-OUTPUT CURVE FOR MOSS LANDING UNIT 7 IN INCREMENTS of 0.1

Following the same procedure of Table 2.1 and Figure 2.7, we plot the Table 2.2 which results to Figure 2.8

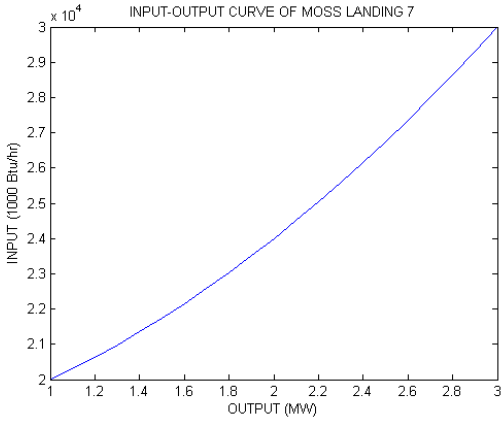


Figure 2.8: INPUT-OUTPUT CURVE of MOSS LANDING 7 [26]

From Figure 2.8, it can be seen that the relationship between the input and output takes the form of a second order polynomial function and thus the initial equation for the fuel cost is given by a quadratic function:

$$f(i,t) = a \cdot P_s(i,t) + b \cdot P_s(i,t) + c \cdot P_s(i,t)^2 \tag{2.3}$$

where a, b, and c are fuel cost coefficients. In finding the proper coefficients, curve fitting from MATLAB is used.

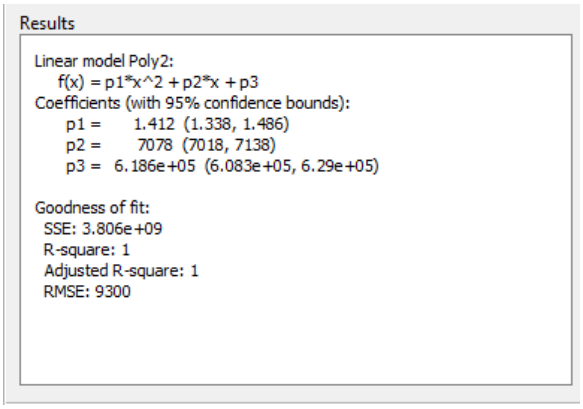


Figure 2.9: Fuel cost coefficients of Moss Landing 7 from Table 2.1 produced by MATLAB

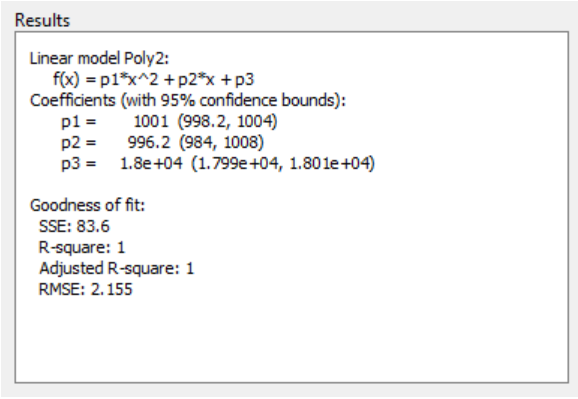


Figure 2.10: Fuel cost coefficients of Moss Landing 7 from Table 2.2 produced by MATLAB

where $p1 = a$, $p2 = b$, and $p3 = c$.

The method of getting the coefficients can be replicated and applied to other thermal plants.

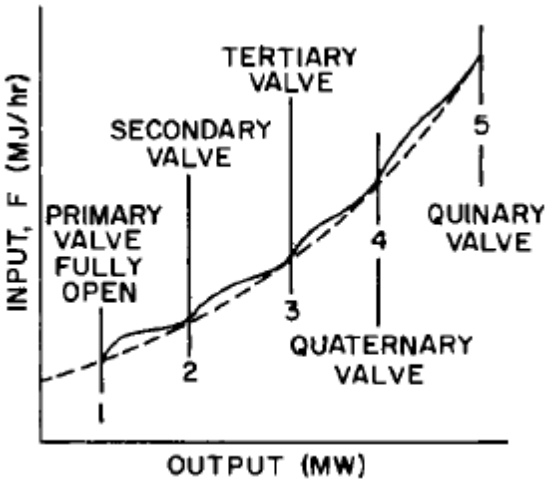


Figure 2.11: valve-point effects as described in [33]

Considering the valve-point effect of thermal plants as can be seen in Figure 2.11, multiple steam valves are opened [15]. The secondary valve is not opened until the primary valve is not fully opened to ensure optimal build up and release of steam. The opening of the multiple valves cannot be modeled by a simple quadratic function and thus the behavior must be added to the computation of fuel cost function (2.3). The opening of the valves can result into a periodic addition of fuel for it to throttle to reach its maximum and reach the minimum of the next valve.

As we can see from the graph of Figure 2.7 and Figure 2.8, the graph in Figure 2.8 have the curve of a quadratic equation while the graph in Figure 2.7 ressembles a linear equation. The deviation of Figure 2.8 to Figure 2.7 is caused by having large intervals and having insufficient data points. The blocks from Table 2.1 represent the point where the valves will open. Instead of the graph of Figure 2.7 being quadratic, it took the form of graph in Figure 2.11. The main reason for the increasing and decreasing values and the deviation to the supposed linear nature of the curve described in Figure 2.11 is because of the throttling of the valve due to high gaseous friction around the valve edges just as the valve is opened and fall off as the valve entrance largens and the pressure flow

smoothens and normalizes. The phenomena is known as the valve-point effect. The behavior of the valve-point effect can be expressed by:

$$f_1(P_s(1,1)) = \begin{cases} a_1(P_s(1,1))^2 + b_1(P_s(1,1)) + c_1 & P_1^{min} < P_s(1,1) < P_1^{max} \\ a_2(P_s(1,1))^2 + b_2(P_s(1,1)) + c_2 & P_2^{min} < P_s(1,1) < P_2^{max} \\ \vdots & \\ a_k(P_s(1,1))^2 + b_k(P_s(1,1)) + c_k & P_k^{min} < P_s(1,1) < P_k^{max} \end{cases} \tag{2.4}$$

where k is the valve region/index, P^{min} and P^{max} are the minimum and maximum power generation limits, respectively. a_k, b_k, and c_k are the fuel cost coefficient for the k^{th} valve region.

The piecewise fuel cost equation 2.4 can be satisfactory but it does not emulate the behavior of smoothing pressure flow. The addition of periodic behavior must be represented by periodic functions specifically in the modeling of the fuel cost function, thus we use the sine function. The sine function that will be added to the fuel cost function is given by:

$$|d_{si} sin(e_{si} g(P_s))| \tag{2.5}$$

Adding 2.5 to 2.3, we get

$$F_{totalcost} = \sum_{t=1}^{T} \sum_{i=1}^{N_s} \left[\left(a_{si} + b_{si} P_s(i,t) + c_{si} (P_s(i,t))^2 \right) + |d_{si} \sin(e_{si} (P_s(i)^{min} - P_s(i,t)))| \right] \tag{2.6}$$

d_{si} and e_{si} are valve point effects coefficients of the i^{th} thermal unit in t time interval. Since many thermal plant systems implements valve points only in two states, which is the open and close state, the opening and closing of these valves affect the conversion of fuel to energy in a sine wave like form as seen from Figure 2.11.

Parameters

$F_{totalcost}$ is the total cost of all the thermal plant of an individual at time $t = 0$ to T
T is the number of time interval measured and incremented per hour
N_s is the number of thermal plant
$P_s(i,t)$ is the power generated by the ith thermal plant at time t

$a_{si}, b_{si}, c_{si}, d_{si}, e_{si}$ are cost coeffcient of the fuel cost function
$P_s(i)^{min}$ and $P_s(i)^{max}$ are the minimum and maximum power generation of the ith thermal plant

Equations

$$F_{totalcost} = \sum_{t=1}^{T} \sum_{i=1}^{N_s} \left[\left(a_{si} + b_{si} P_s(i,t) + c_{si} (P_s(i,t))^2 \right) + |d_{si} \sin(e_{si}(P_s(i)^{min} - P_s(i,t)))| \right] \tag{2.7}$$

Equality and Inequality Constraints

1. Generation Limits

$$P_s(i)^{min} \leq P_s(i,t) \leq P_s(i)^{max} \tag{2.8}$$

2. Power System Balance

$$\sum_{i=1}^{N_s} P_s(i,t) + \sum_{j=1}^{N_h} P_h(j,t) = P_{D(t)} + P_{L(t)} \tag{2.9}$$

where $P_{D(t)}$ is the power demand and $P_{L(t)}$ is the transmission loss at the corresponding time.

Constraints Handling Method

The constraints handling is applied to the power generation of the thermal plant to satisfy the power demand. Given the power generation of the hydro plant, the constraints handling method uses the power generation of the thermal plant to fill the gap left by the power generated by the hydro plant to a given power demand. Having the power deficit $\Delta P(t)$, the constraints handling method distributes the power deficit $\Delta P(t)$ to all the thermal plant available. After distributing the power deficit to all the thermal plants, the constraints handling method will find the ith maximum adjustable thermal plant and the ith thermal plant will shoulder the remaining deficit. The constraints handling method

Algorithm 2.1.1: CONSTRAINTS HANDLING FOR THERMAL PLANTS()

is given by:

$$
\begin{array}{l}
\textbf{for } t \leftarrow 1 \textbf{ to } T \\
\quad \textbf{do} \begin{cases}
iteration \leftarrow 0 \\
\Delta P(t) = P_D(t) + P_L(t) - \sum_{i=1}^{N_s} P_s(i,t) - \sum_{j=1}^{N_h} P_h(j,t) \\
\textbf{while } |\Delta P(t)| > \varepsilon_{P_{course}} \textit{ and } iteration \leq iteration_course \\
\quad \textbf{do} \begin{cases}
averageP(t) = \Delta P(t)/N_s \\
\textbf{for } i \leftarrow 1 \textbf{ to } N_s \\
\quad \textbf{do} \begin{cases} P_s(i,t) = P_s(i,t) + averageP(t) \\ adjust\_thermal() \end{cases} \\
\Delta P(t) = P_D(t) + P_L(t) - \sum_{i=1}^{N_s} P_s(i,t) - \sum_{j=1}^{N_h} P_h(j,t) \\
iteration++
\end{cases} \\
iteration \leftarrow 0 \\
\textbf{while } |\Delta P(t)| > \varepsilon_{P_{fine}} \textit{ and } iteration \leq iteration_fine \\
\quad \textbf{do} \begin{cases}
Calculate\ \Delta P_s(i,t)\ from\ i = 1\ to\ N_s \\
Select\ r\ with\ the\ maximum\ adjustable\ generation \\
P_s(r,t) = P_s(r,t) + \Delta P(t) \\
adjust\_thermal() \\
\Delta P(t) = P_D(t) + P_L(t) - \sum_{i=1}^{N_s} P_s(i,t) - \sum_{j=1}^{N_h} P_h(j,t) \\
iteration++
\end{cases}
\end{cases}
\end{array}
$$

2.2 Hydro Plant

Hydro plant is a system of machines and components in which it utilizes gravitational force and converts the mechanical energy from the downward flow of water to generate electricity. A hydro plant usually have a reservoir, water intake, penstock, turbine, and powerhouse that houses the electric generator.

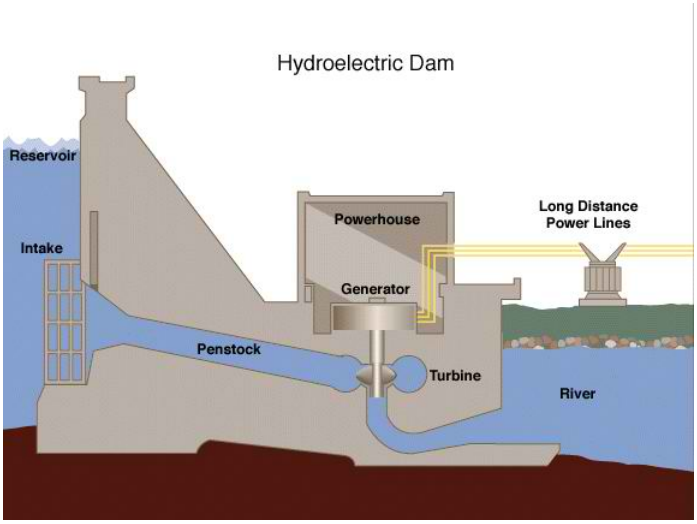


Figure 2.12: An overview of the major parts of a hydro plant namely the water reservoir, water intake, penstock, turbine, powerhouse, and electric generator [3]

2.2.1 Reservoir

The reservoir stores the water needed to produce electricity. Reservoirs usually have a source of water where it replenishes its stored water. The inflow source of water usually comes from rivers and rainfall. Reservoirs are located higher than the rest of the major parts. The higher the water reservoir is with respect to the turbine, the higher the potential energy that can be converted to kinetic energy.

2.2.2 Water Intake/Control Gates

These huge control gates control the flow of water from the reservoir to the penstock.

2.2.3 Penstock

The penstock is a long pipe/shaft from the reservoir to the turbine. The configuration of the penstock can also contribute to the overall efficiency of the hydro plant. The smoother and higher the angle of the penstock is with respect to the ground, the higher the kinetic energy of the water is.

2.2.4 Water Turbines

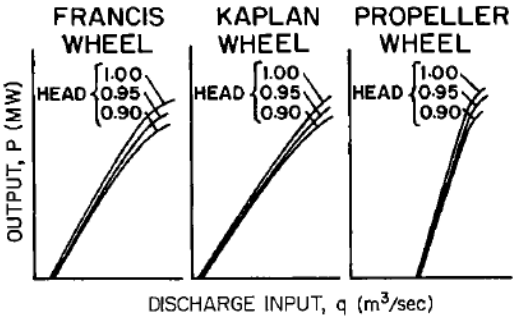


Figure 2.13: Different types of water turbines and their general efficiency using the input-output curve [33]

Figure 2.14: Different types of water turbines. From left to right: Pelton Wheel, two types of Francis turbine, and Kaplan turbine [4]

Water turbines convert the kinetic energy of the falling water into rotational mechanical energy. Because of gravity, the falling water that came from the higher position of the water reservoir travels through the penstock and through the blades of the water turbines. The water turbines rotate as the water flows through them because of the angle of the blades. The water turbines are connected through a metal bar wherein the mechanical energy is transfered to the to the electric generators.

2.2.5 Electric Generator

The electric generator does not fully deviate on the design of the electric generator found in thermal plants. It employs the same concept of generating electricity by magnetic induction.

2.2.6 Mathematical Model

Power Generation

A well established functional[19, 43] for the power generation of hydro plants is given by:

$$P_h(j,t) = C_{1j}(V_h(j,t))^2 + C_{2j}(Q_h(j,t))^2 + C_{3j}V_h(j,t)Q_h(j,t) + C_{4j}V_h(j,t) + C_{5j}Q_h(j,t) + C_{6j} \tag{2.10}$$

where $C_{1j}, C_{2j}, C_{3j}, C_{4j}, C_{5j}$, and C_{6j} are generation coefficient of the jth hydro plant, $V_h(j,t)$ is the volume and $Q_h(j,t)$ is the water discharge of the j^{th} reservoir in time interval t. There are other variations of getting $P_h(j,t)$ in [43, 33, 39, 34, 14, 8]. The Equation 2.10 was derived from different models[33, 29, 39, 20] to have a more generalized function. In acquiring the Equation 2.10, a well established equation for the output power of hydro-turbo generation given by [33]:

$$P_h = (qh/102)\eta_t\eta_G \tag{2.11}$$

where q is the water discharge and h is the effective net head. η_t is the turbine efficiency and η_G is the generator efficiency.

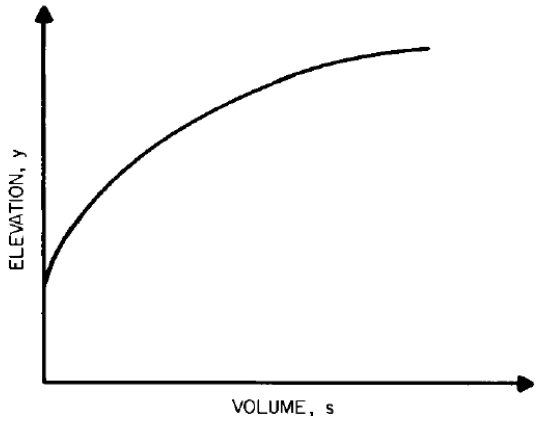


Figure 2.15: A typical elevation-storage curve which takes the form of an inverted second order polynomial function [33]

We can approximate the elevation and storage curve using a second-order polynomial function:

$$y = \alpha_0 + \alpha_1 s + \alpha_2 s^2 \tag{2.12}$$

based on Figure 2.15. y is the elavation and s is the volume.

The effective net head h is equal to the difference between the gross head h_g and the head loss of the penstock,h_p:

$$h = h_g - h_p \tag{2.13}$$

The gross head h_g is equal to the difference between elevation y and the tailrace elevation y_τ

$$h_g = y - y_\tau \tag{2.14}$$

The tailrace elevation $y_\tau$is a function of water discharge q and spillage σ. The relationship between the two variables is linear and can be expressed by:

$$y_\tau = y_{tau_0} + B_\tau(q + \sigma) \tag{2.15}$$

where B_τ is the tailrace elevation coeffcient. The head losses in the penstock h_p is:

$$h_p = h_{p_0} + B_p q \tag{2.16}$$

where B_p is the head loss coeffcient. The resulting expression for the effective net head h is equal to:

$$h(t) = y(t) - [(y\tau_0 + h_{p_0}) + B_\tau\sigma(t) + (B_\tau + B_p)q(t)] \tag{2.17}$$

In modelling the reservoir, where volume s and the shape of the reservoir affects the elevation of water surface y and the water discharge rate q.

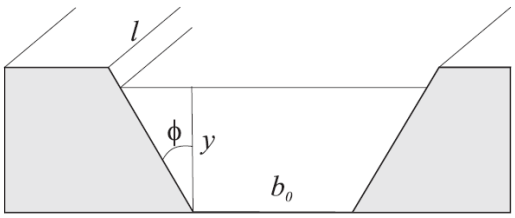


Figure 2.16: Trapezoidal reservoir configuration [29]

Most of reservoirs are shaped like a trapezoid. Based on Figure 2.16, we get

$$s(t) = b_0 \cdot l \cdot y(t) + l \cdot \tan\phi \cdot y^2(t) \tag{2.18}$$

Getting P_h using the equations above, we get:

$$P_h = q(t)[(y - y_{\tau_0} + B_\tau(q + \sigma)) - h_{p_0} + B_p q] \tag{2.19}$$

if we use

$$y = y_0 + B_y s + C_y s^2 \tag{2.20}$$

which is the second order polynomial approximation of the elevation, we get

$$P_h = q[((y_0 + B_y s + C_y s^2) - y\tau_0 + B_\tau(q+\sigma)) - h_{p_0} + B_p q] = B_\tau B_p q^2 + C_y q s^2 + y_0 y_{\tau_0} B_\tau \sigma h_{p_0} q \tag{2.21}$$

Since P_h is a function of water discharge rate and h, where s and h have a linear relationship. We can substitute s in place of h. P_h can be simplified to

$$P_h = C_1 q^2 + C_2 q s^2 + C_3 q s + C_4 q \tag{2.22}$$

where C_1, C_2, C_3, and C_4 are the coefficients. The same method was used in [29]. Considering other hydro-reservoir model, an exponential variation of hydro power generation is the Arvanitidis-Rosing model given by:

$$P_h = qh(\beta - e^{-\alpha s}) \tag{2.23}$$

From Figure 2.15, equation 2.22 and 2.23, the dependency of s in Equation 2.23 to water discharge q shown in equation 2.17 if

$$y(t) = -h(t) - [(y\tau_0 + h_{p_0}) + B_\tau \sigma(t) + (B_\tau + B_p)q(t)] \tag{2.24}$$

and the dependency of water discharge q to the effective net head h shown in Equation 2.17, it can be deduced that the hydro power generation follows the exponential function of Arvanitidis-Rosing Model equation 2.23 and the previously derived equation 2.22. Connecting the Arvanitidis-Rosing Model and Equation 2.23, we use quadratic approximation in multiple variables from the Taylor series:

$$\begin{aligned} f(x,y) \approx f(a,b) + f_x(a,b)(x-a) + f_y(a,b)(y-b) + \frac{1}{2!}[f_{xx}(a,b)(x-a)^2 + \\ 2f_{xy}(a,b)(x-a)(y-b) + f_{yy}(y-b)^2] \end{aligned} \tag{2.25}$$

where $x \approx q$ and $y \approx h$. $f(a,b)$, $f_x(a,b)$, $f_y(a,b)$, $f_{xx}(a,b)$, $f_{yy}(a,b)$, $f_{xy}(a,b)$, a, and b will dissolve into constants. Simplifying the equation 2.25 by getting the general structure of the equation:

$$f(x,y) \approx C_1 \cdot x^2 + C_2 \cdot y^2 + C_3 \cdot x \cdot y + C_4 \cdot x + C_5 \cdot y + C_6 \tag{2.26}$$

From Equation 2.26 and 2.22

$$f(x,y) \approx \overbrace{C_1 \cdot q^2 + C_3 \cdot q \cdot h + C_4 \cdot q}^{\text{same structure}} + C_2 \cdot h^2 + C_5 \cdot h + C_6 \tag{2.27}$$

$$P_h = \overbrace{C_1 \cdot q^2 + C_3 \cdot qh + C_4 \cdot q}^{\text{same structure}} + C_2 q h^2 \tag{2.28}$$

Having the same structure as can be seen in Equation 2.27 and 2.28, with only minor differences, we can use the Equation 2.26 as the generalized form of P_h, thus we have arrived to equation 2.10.

Parameters

$Q_h(j,t)$ is the water discharge rate of the jth hydro plant at time t
$V_h(j,t)$ is the reservoir water volume of the jth hydro plant at time t
$P_h(j,t)$ is the power generated by the jth hydro plant at time t
$I_h(j,t)$ is the water inflow rate of the jth hydro plant's reservoir at time t
$S_h(j,t)$ is the water spillage rate of the jth hydro plant's reservoir at time t
R_j is the number of upstream hydro plant reservoir of the jth hydro plant
$\tau_{k,j}$ is the water transport time delay from kth hydro plant to the jth hydro plant
$Q_h(j)^{min}$ and $Q_h(j)^{max}$ are the minimum and maximum water discharge rate of the jth hydro plant, respectively
$V_h(j)^{min}$ and $V_h(j)^{max}$ are the minimum and maximum reservoir storage volume limit of the jth hydro plant, respectively
$P_h(j)^{min}$ and $P_h(j)^{max}$ are the minimum and maximum power generation of the jth hydro plant
$V_h(j)^{begin}$ and $V_h(j)^{end}$ are the initial and final storage volume of the jth hydro plant

Water Volume Balance

A hydro plant consists of water reservoir for every powerhouse as shown in Figure 2.17.

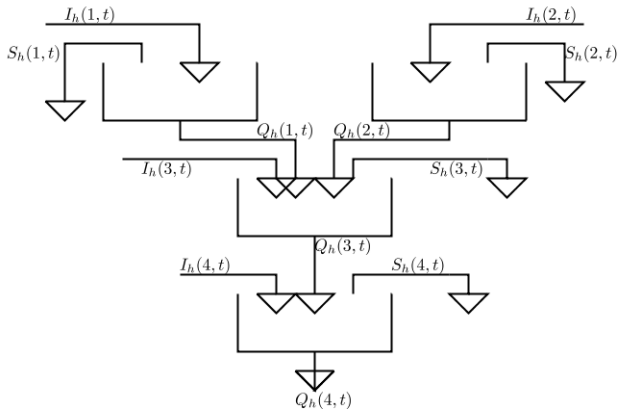


Figure 2.17: Hydro Plant Schematics. A representation of reservoir with their own powerhouse, inflow rate $I_h(j,t)$, spillage rate $S_h(j,t)$ and water discharge rate $Q_h(j,t)$.

From the set of cascaded water reservoir of power plants, we can derive an equation for $V_h(j,t)$ for a jth hydro plant's reservoir at time interval t.

$$V_h(j,t) = V_h(j,t-1)+I_h(j,t)-Q_h(j,t)-S_h(j,t)+\sum_{k=1}^{R_{uj}}[Q_h(k,t-\tau_{kj})+S_h(k,t-\tau_{kj})] \quad (2.29)$$

Constraints

1. Generation Limits

$$P_h(j)^{min} \leq P_h(j,t) \leq P_h(j)^{max} \quad (2.30)$$

2. Water Discharge Rate Limits

$$Q_h(j)^{min} \leq Q_h(j,t) \leq Q_h(j)^{max} \quad (2.31)$$

3. Reservoir Storage Volume Limits

$$V_h(j)^{min} \leq V_h(j,t) \leq V_h(j)^{max} \quad (2.32)$$

4. Initial and Terminal Reservoir Storage Volumes

$$V_h(j,t)|^{t=0} = V_h(j)^{begin} V_h(j,t)|^{t=T} = V_h(j)^{end} \tag{2.33}$$

Constraints Handling Method

The constraints handling method makes use of the equality and inequality constraints imposed by the decision maker and the system.

Constraints handling is important for the SHS to work effectively with respect to the limits of the hydrothermal system. The method sets the bounds and limits of every value of the solution set and its corresponding equality and inequality constraints. In the equality constraints handling of the hydro plants, water volume reservoir is the main concern since every hydro plant cannot produce power if it does not have enough water to move the water turbines. The water discharge load $\Delta V_h(j)$ acquired from the violation of the end water reservoir and desired end water reservoir should be distributed to all time intervals to prevent the depletion of water and to meet the terminal water reservoir storage state. After distributing the load equally to every water reservoir, the method distributes the remaining water discharge load to the water reservoir with the most adjustable value with respect to its water reservoir storage volume limits and the terminal water reservoir storage value. The process can be showed by the pseudocode:

Algorithm 2.2.1: CONSTRAINTS HANDLING FOR HYDRO PLANTS()

volume_constraints_generation()
for $j \leftarrow 0$ **to** N_s
do
- $iteration \leftarrow 0$
- $adjust\_discharge(Q_h(j,t))$
- $update\_volume\_constraints()$
- $\Delta V_h(j) = V_h(j,T) - V_h(j)^{end}$
- **while** $|\Delta V_h(j)| > \varepsilon_{v_{course}}$ *and* $iteration \leq iteration_{course}$
  - **do**
    - $averageV_h(j) = \Delta V_h(j)/T$
    - **for** $t \leftarrow 1$ **to** T
      - **do**
        - $Q_h(j,t)Q_h(j,t) + averageV_h(j)$
        - $adjust\_discharge(Q_h(j,t))$
    - $\Delta V_h(j) = V_h(j,T) - V_h(j)^{end}$
- $iteration \leftarrow 0$
- **while** $|\Delta V_h(j)| > \varepsilon_{v_{fine}}$ *and* $iteration \leq iteration_{fine}$
  - **do**
    - *Calculate* $\Delta Q_h(j,t) from t = 1 to T$
    - *Select k with the maximum adjustable discharge*
    - $Q_h(j,k) = Q_h(j,k) + \Delta V_h(j)$
    - $adjust\_discharge(Q_h(j,k))$
    - $iteration++$

2.3 Short-term Hydrothermal Scheduling(SHS)

Short-term hydrothermal scheduling is a complicated, nonlinear, dynamic, non-convex, multi-objective optimization problem. It is significant in the economic operation of connected thermal power systems with hydro plants. The primary objective of SHS optimization is to minimize the fuel cost consumption of the thermal plants while maximizing the use of hydro plants while satisfying many equality and inequality constraints. Equality and inequality constraints include hydraulic constraints, reservoir water volume, power balance constraints, thermal generation limits, and water discharge rate limits for hydro plants.

2.3.1 Data Representation and Decision Variables

The decision variables are the numerical quantities that the algorithm controls. With the given mathematical model above and the nature of data representation needed for the methods, an individual of a population consists of set of solutions for the water discharge rate and the power generation of the thermal plants. Given a population of individual solutions, these solutions should be subjected to their respective constraints such as the power generated by thermal and hydro plants should be equal to the power demand and power loss. The water discharge rate should also satisfy the water volume of every reservoir by limiting the water discharge in every time interval t. They are commonly denoted as a vector $\vec{X_n}$ given by:

$$X_n = \begin{bmatrix} Q_h(1,1) & Q_h(2,1) & \dots & Q_h(N_h,1) & P_s(1,1) & P_s(2,1) & \dots & P_s(N_s,1) \\ Q_h(1,2) & Q_h(2,2) & \dots & Q_h(N_h,2) & P_s(1,2) & P_s(2,2) & \dots & P_s(N_s,2) \\ \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \dots & \vdots \\ Q_h(1,T) & Q_h(2,T) & \dots & Q_h(N_h,T) & P_s(1,1) & P_s(2,1) & \dots & P_s(N_h,T) \end{bmatrix} \tag{2.34}$$

where the vector $\vec{X}$ is an individual in a population. An individual will have a fitness value from the fitness function. A fitness value is denoted by F_n, where n is the nth individual in the population. The whole population is denoted by:

$$X = [X_1, X_2 \dots, X_N]$$

where N is the number of individuals in a population.

Having many individuals in the population, the fitness values of every individual is stored in a vector denoted by F_n:

$$F = [F_1, F_2, \dots, F_N]$$

The fitness value of F_1 corresponds to the fitness value of X_1, F_2 to X_2 and so on.

2.3.2 Initialization of Population

The initial population will be generated by:

$$\begin{aligned} Q_h(j,t) &= Q_h(j)^{min} + U(0,1) \times \left(Q_h(j)^{max} - Q_h(j)^{min}\right) \\ P_s(i,t) &= P_s(i)^{min} + U(0,1) \times \left(P_s(i)^{max} - P_s(i)^{min}\right) \end{aligned} \tag{2.35}$$

where U is a randomly generated number from 0 to 1 exclusively. $Q_h(j)^{max}$ and $Q_h(j)^{min}$ are the upper and lower limits of the water discharge rate$P_s(i)^{min}$ and $P_s(i)^{min}$ are the upper and lower limits of the power generation of the ith thermal plant.

The equation 2.35 will fill an individual 2.34.

The fitness function is the measurement on how good an individual is. The fitness function generates an output value from a set of input solutions from the population. A well composed objective function should gauge all the individuals in a population well. The fitness function is the objective function of the SHS problem 2.7.

2.4 Genetic Algorithm(GA)

Genetic algorithm is an evolutionary algorithm based on Darwin's survival of the fittest and natural selection through the genetic make up of an individual [23]. It was first proposed by John Holland in the 1970s through his book *Adaptation in Natural and Artificial Systems (1975)* and is used primarily in optimization and search problems. The classical genetic algorithm is encoded in binary due to the nature of the genes encoded only in 4 states, namely A,T,G, and C. A general flow of a classical GA process is shown in Figure 2.18.

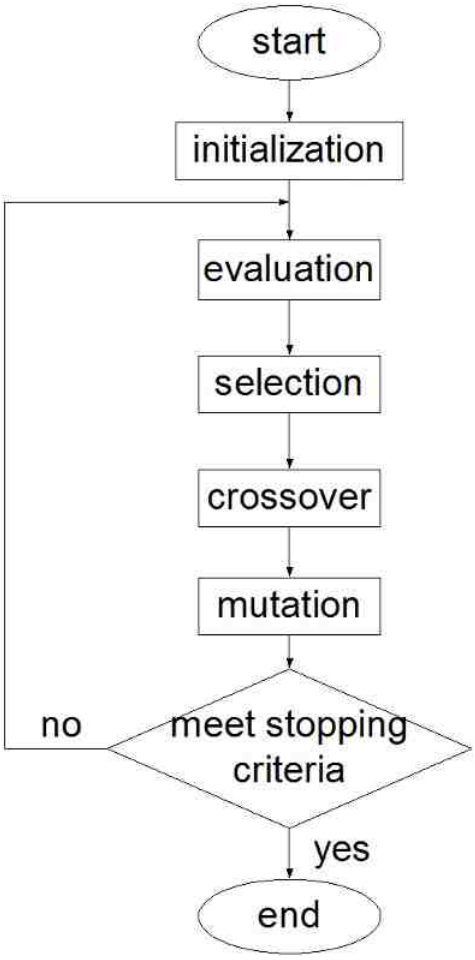


Figure 2.18: Generalized flow of the GA process

2.4.1 Operators of GA

Evaluation

The evaluation process is analogous to the acquisition of fitness values of every individual through the use of the fitness function. Every individual in the population have their fitness values where the criteria for different selection operator is based.

Selection

The selection method choose two parents to produce two offsprings. Variants of the selection method is expounded in Section 2.5.1.

Crossover

The crossover method produces offsprings from two or more parents through the concept of reproduction. The reproduction of two or more parents results to offsprings that have inherited their parents' genes. Different traditional types of crossover are:

1. One-point Crossover

   A single point on the genes of two parents is selected either randomly or using different methods. After choosing a point on the two parents, all the genes beyond the parents are interchanged. The new individuals are the offsprings. An example is if the crossover point is 2 and we are given two parents with their genes:

   $$P_1 = 0110111$$

   $$P_2 = 1110011$$

   the resulting children will be

   $$C_1 = 0110011$$

   $$C_2 = 1110111$$

2. Two-Point Crossover

   The same concept of the one-point crossover is used in the two-point crossover. The difference between the one-point crossover and two-point crossover is that the two-point crossover uses two points of crossover to promote the preservation of the parents' genes. Reducing the disruptive process of the crossover operator can contribute to finding a better solution. An example is if the crossover point is 2 and 4, and using the same parent from one-point crossover

   $$P_1 = 0110111$$

   $$P_2 = 1110011$$

   the resulting children will be

   $$C_1 = 0110111$$

   $$C_2 = 1110011$$

   Different variants

3. Cut-and-Splice

   The cut-and-splice crossover choose a single point crossover for the first parent and another sinple point crossover for the second parent. The genes are then exchange between the crossover points and may result to unequal lengths of genes. An example of cut-and-splice crossover is if we have the first crossover point for the first parent as 2, and the second crossover point for the second parent as 5, using the same parent from one-point crossover

$$P_1 = 0110111$$

$$P_2 = 1110011$$

   the resulting children will be

$$C_1 = 0111$$

$$C_2 = 1110010111$$

Mutation

The mutation method mutates individuals to diversify the whole population. In the classical genetic algorithm, individuals encoded in binary can have their values of 1's and 0's to be flipped or changed depending on the predisposition of the decision maker. An example is if we have an individual

$$P_1 = 0110111$$

and the third gene is mutated and flipped from 1 to 0, the resulting mutated individual will be

$$P_1 = 0100111$$

There are many variants of GA that were developed to cater to the needs of some field. In mathematical modeling and optimization of complex systems, a usual variant of GA which is the real coded genetic algorithm is always used.

2.5 Real Coded Genetic Algorithm(RCGA)

2.5.1 Operators of RCGA

As shown in Figure 2.18, the main components of the GA process includes the initialization and evaluation of the population, selection of the fittest individuals, crossover or the reproduction of the individuals, and mutation of the population. The process from the evaluation down to the mutation operator is iterated until a stopping criteria is met. The stopping criteria is usually based on the redundancy of the most fit individual to the population or a set number of iterations.

The difference between the classical GA and the RCGA is the way data is represented. In GA, it is usually in binary form while in RCGA, the data is encoded in real values. A good fitness function and data representation are also crucial in the GA process.

Selection

The selection operator selects fit individuals to be carried over to the next generation. Many types of selection are available for use and is discussed in [7]. The selection operator determines the convergence characteristic of RCGA. The chosen selection method for solving SHS problem is elitist selection due to its simplicity and the ability to preserve the fittest individuals and balance the disruptive nature of the crossover and mutation operators. Different types of selection are:

1. Tournament Selection

   Tournament selection randomly chooses atleast two individuals in the population and compare their fitness value. The best individuals for every tournament will be carried over to the next operators. The process will be done until a set number of individuals is satisfied. For example, given atleast two individuals X_1 and X_2,

where X_1 is given by:

$$X_1 = \begin{bmatrix} 6.286 & 10.888 & 17.728 & 19.139 & 614.473 \\ 7.025 & 6.495 & 24.228 & 13.916 & 1408.858 \\ 9.248 & 8.603 & 19.135 & 19.220 & 508.199 \\ 10.853 & 13.106 & 20.974 & 15.026 & 1306.477 \\ 5.048 & 12.754 & 10.884 & 15.423 & 2140.580 \\ 10.289 & 12.332 & 23.131 & 15.012 & 1802.334 \\ 5.801 & 12.407 & 21.384 & 16.137 & 1298.278 \\ 11.635 & 6.186 & 26.206 & 16.807 & 1144.574 \\ 10.374 & 8.144 & 16.826 & 14.621 & 1117.317 \\ 5.792 & 10.457 & 27.300 & 15.581 & 1603.367 \\ 9.368 & 13.049 & 13.471 & 16.530 & 1888.533 \\ 13.561 & 13.268 & 24.568 & 20.491 & 2116.780 \\ 8.961 & 7.472 & 14.896 & 18.748 & 2169.661 \\ 10.806 & 11.287 & 10.917 & 19.594 & 732.386 \\ 7.481 & 10.335 & 23.684 & 21.219 & 2366.600 \\ 8.263 & 13.933 & 15.528 & 19.541 & 645.905 \\ 13.098 & 8.009 & 19.400 & 13.771 & 1479.827 \\ 12.416 & 13.345 & 21.114 & 16.716 & 2099.820 \\ 7.450 & 13.526 & 15.376 & 14.729 & 949.400 \\ 13.743 & 14.092 & 23.235 & 24.775 & 2124.337 \\ 14.890 & 10.207 & 28.781 & 18.824 & 540.201 \\ 14.325 & 8.338 & 21.022 & 13.481 & 1462.222 \\ 9.930 & 7.664 & 16.247 & 16.139 & 2355.188 \\ 12.175 & 13.470 & 23.892 & 22.326 & 995.791 \end{bmatrix}$$

and X_2 is given by:

$$X_1 = \begin{bmatrix} 11.029 & 14.590 & 20.913 & 18.435 & 2367.554 \\ 6.945 & 14.904 & 17.823 & 14.951 & 843.229 \\ 13.739 & 12.203 & 16.937 & 15.848 & 1560.784 \\ 9.584 & 6.457 & 18.428 & 21.616 & 645.339 \\ 8.758 & 6.801 & 11.750 & 13.180 & 1797.854 \\ 9.599 & 10.068 & 11.799 & 13.381 & 743.784 \\ 11.605 & 8.069 & 29.751 & 17.333 & 1567.760 \\ 5.559 & 14.902 & 11.454 & 20.573 & 1165.882 \\ 9.196 & 13.275 & 24.049 & 21.128 & 2419.106 \\ 11.722 & 12.127 & 12.098 & 18.943 & 1192.839 \\ 14.036 & 13.467 & 20.072 & 17.896 & 690.038 \\ 10.966 & 10.010 & 24.738 & 13.390 & 764.838 \\ 5.177 & 9.418 & 20.262 & 14.840 & 1740.942 \\ 14.890 & 9.785 & 16.081 & 21.529 & 1786.196 \\ 8.305 & 14.386 & 15.058 & 14.497 & 732.893 \\ 11.078 & 12.722 & 22.472 & 13.358 & 1162.426 \\ 10.711 & 8.450 & 12.105 & 20.828 & 679.517 \\ 10.679 & 10.678 & 26.969 & 15.521 & 1828.142 \\ 6.982 & 10.907 & 21.347 & 15.855 & 2156.879 \\ 11.359 & 8.677 & 11.832 & 16.594 & 2230.526 \\ 6.477 & 14.409 & 17.057 & 19.113 & 846.753 \\ 6.668 & 10.797 & 23.860 & 13.934 & 1077.695 \\ 11.273 & 10.137 & 18.884 & 22.505 & 2024.068 \\ 8.877 & 11.827 & 10.517 & 14.846 & 1532.053 \end{bmatrix}$$

If their fitness value is 908052.616 and 874238.213, respectively, the method will select X_2 since its fitness value is lower and the problem is a minimization problem. The method will be iterated until the required number of individuals are met.

2. Truncation Selection

   In truncation selection, only the best subsets of the whole population are candidates

for selection. The best subset where all the fittest individuals are then subjected to other operators based on their fitness. The probability of undergoing through the operators stays the same to all the individuals of the best subset. The selection method is done by sorting all the individuals and having a threshold N_B. The threshold N_B will be the divider to the best subset and the rest. The subset where the worst individuals are included is truncated. An example of truncation selection is when we have 100 individuals in the population N_P, and the required number of individuals to be selected N_S is equal to 50, and $N_B = 60$, we truncate preserve the fittest N_B number of individuals and discard the rest. From The N_B number of individuals, we choose 50 N_S individuals with equal probability.

3. Linear Ranking Selection

   All the individuals are ranked and sorted from worst to best. The probability of being selected is based on the ranking and their fitness value. The higher the number of the rank and the better the fitness value, the higher the probability of an individual to be selected. The probability is given by:

$$P_i = \frac{1}{N}(\eta + (\eta^+ - \eta^- \frac{i-1}{N-1})$$

   where

$$i\epsilon\{1, \ldots, N\}$$

   is the ith individual. η is the selection bias parameter. The higher the value of η, the more it will favor the best individuals in the population. The common pitfalls of using a high value of η is the stagnation of the individuals since after many generations, almost the same set of individuals are used. η^- and η^+ are given by:

$$\eta^- = \frac{2}{r_m + 1}$$

$$\eta^+ = \frac{2r_m}{r_m + 1}$$

   where r_m is the multiplication factor which determines the gradient of the linear function. An example of linear ranking selection is when we have 6 individuals with their fitness values, respectively is given by:

$$F = [5, 10, 20, 21, 23, 24]$$

If we have $r_m = 0.1$, then $\eta^- = 1.818$ and $\eta^+ = 0.181$. The probability will be:

$$P_N = [0.0301667, 0.0756389, 0.121111, 0.166583, 0.212056, 0.257528]$$

respective to the position of the individual and in P_N.

4. Elitist Selection

   The elitist selection method preserves the subset of the best population. The selected subset is already the population that will undergo crossover and mutation. The method is done to preserve the best known solutions/individuals. Elitist selection is commonly used in combination with other selection strategies.

5. Proportional Selection

   Proportional selection is the first method to be proposed to be used with the classical GA developed by Holland. The probability of an individual being selected is based on its fitness value and relation to others' fitness value. The probability of an individual is given by:

   $$P_i = \frac{f_i}{M}$$

   where M is the summation of all the fitness value in the population. As an example, assume that a population of 6 individuals with fitness values:

   $$F = [5, 10, 11, 1, 2, 6]$$

   Undergoing the probability function, $M = 35$ and

   $$P_N = [0.1429, 0.2857, 0.3143, 0.0286, 0.0571, 0.1714]$$

   respective to the position of P_N and F, where N is the number of population.

Crossover

Crossover is the process wherein two parents selected from the population will produce two children individuals with interchanged part of the two parents. In BCGA, a crossover point is usually selected at random. The left chromosome of the first parent is then interchanged with the right chromosome of the second parent to produce one offspring.

The same is done vice-versa to produce another offspring.In the crossover operator, simulated binary crossover (SBX) is used and as the name implies, it mimics the behavior of a crossover point for the values encoded in binary if given real values. SBX uses a probability distribution around two parents to create two children individuals. Since real coded values are used and not binary coded values, SBX is prefered than using binary crossover operator to avoid the complexity of converting good and precise real values to binary code. The process of reproduction of two children, $Y^(i) = (y_1)^{(1)}, y_2^{(1)}, \ldots, y_n^{(1)})$ and $Y^{(2)} = (y_1^{(2)}, y_2^{(2)}, \ldots, y_n^{(2)})$ from the parent individuals, $X^(i) = (x_1)^{(1)}, x_2^{(1)}, \ldots, x_n^{(1)})$ and $X^{(2)} = (x_1^{(2)}, x_2^{(2)}, \ldots, x_n^{(2)})$ follows the simulated binary crossover (SBX) proposed by Kalyanmoy and Agrawal. The production of two children individuals are given by the equation:

$$y_i^{(1)} = 0.5[(x_i^{(1)} + x_i^{(2)}) - \gamma|x_i^{(2)} - x_i^{(1)}|]y_i^{(2)} = 0.5[(x_i^{(1)} + x_i^{(2)}) + \gamma|x_i^{(2)} - x_i^{(1)}|] \quad (2.36)$$

where

$$\gamma = \begin{cases} (\alpha u)^{1/(\eta_c+1)} & if u \leq \frac{1}{\alpha} \\ \frac{1}{2-\alpha u}^{1/\eta_c+1} & otherwise \end{cases}$$

where $\alpha = 2 - \beta^{-(\eta_c+1)}$ and β is defined as:

$$\beta = 1 + \frac{2}{x_i^{(2)} - x_i^{(1)}} \min[(x_i^{(1)} - x_i^l), (x_i^u - x_i^{(2)})]$$

Mutation

Mutation mutates an individual wherein changing parts of the chromosome is done. The mutation operator is responsible for keeping the individuals diverse to prevent premature convergence. Michalewicz's Non-Uniform Mutation(NUM)[18] is used. An individual is represented as $X = (x_1, x_2, \ldots, x_n)$. The mutated individual will produce $X_i^{'}$ which is defined as

$$X_i^{'} = \begin{cases} x_i + \Delta(t, x_i^u - x_i) & (\tau = 0) \\ x_i - \Delta(t, x_i^u - x_i) & (\tau = 1) \end{cases} \quad (2.37)$$

where τ is a boolean value and t is the current iteration number. The function $\Delta(t, y)$ is given as follows:

$$\Delta(t, y) = y(1 - r^{1-t/t_{max}^b})$$

These newly created individuals will then be carried to the next generation.

2.5.2 Next Generation

All individuals produced in the crossover, the elite subset of the population, and the newly mutated individuals will be carried over to the next generation.

2.5.3 Stopping Criterion

The stopping criterion for the RCGA is based on number of iterations. The process stops when a specified iteration have been met.

2.6 Artificial Fish Swarm Algorithm (AFSA)

Artificial fish swarm algorithm was first proposed by Li [31]. AFSA is used as a local optimizer to improve the population produced by the RCGA algorithm. The artificial fish swarm algorithm uses and mimics the behavior of fishes. In the physical world, fishes often travel in a group known as a school. A school of fish will always explore and find areas that are rich in food and is ideal for breeding and living. An individual fish may follow other fish in hopes of finding a good location that is rich in food and is safe from danger. It can also exhibit a behavior of swarming which makes the school of fish safer from predators and hazards. A fish have a simple behavior, wherein they follow the area with the most consistent food. Because of the exhibited behaviors of fishes, we can use different the different components of the AFSA by simulating fish behaviors to achieve optimal solutions.

There are three main types of behavior enacted by a fish which are the follow behavior, swarm behavior, and prey behavior. Each position held by a fish can be a potential solution for the SHS problem. The variables that should be considered in formulating the AFSA for SHS problem is the current position of every fish, their step length or the maximum distance that a fish can travel in a given interval of time, its visual or the length of its field of sight, the crowd factor or the rating of a particular area inhabited by other fishes, and the number of fishes that is within its visual. The three main functions

are behavior of searching food(prey), behavior of swarm, and behavior of agents(follow) [36].

2.6.1 Components of AFSA

The three main components of the AFSA mimics the behavior of the fish.

Prey Behavior

If a fish discover an area within its visual distance where food is abundant, it will swiftly proceed to the area. Preying is a basic biological behavior wherein the survival of every individual fish is highlighted.The behavior is expressed as follows:

$$Prey(X_i^{t+1}) = \begin{cases} X_i^{(t)} + \frac{X_j - X_i^{(t)}}{||X_j - X_i^{(t)}||}.step.rand() & if y_j < y_i \\ X_i^{(t)} + step.rand() & else \end{cases}$$

Swarm Behavior

A school of fish will tend to swarm to an area where concentration of food is high. The area of high food concentration is usually at the center of the swarm within the visual of the fish. Swarming behavior ensures the survival of the school since every fish is protected by the swarm. In cases wherein predation of other species is imminent, the swarm behavior minimizes the mortality of the school. The current position of a fish is X_i. A swarm will have a center, X_c which is $X_c = \sum_{j=1}^{n_f} \frac{X_j}{n_f}$, where n_f is the number of fishes within X_i visual. Swarm behavior is expressed as:

$$Swarm(X_i^{t+1}) = \begin{cases} X_i^{(t)} + \frac{X_c - X_i^{(t)}}{||X_c - X_i^{(t)}||}.step.rand() & if \frac{y_c}{n_f} < \delta y_i \\ Prey(X_i^{(t+1)}) & else \end{cases}$$

Follow behavior

A fish will follow other fish if the other fish discovers more food. Follow behavior is closely related to the swarm behavior but differs in the point where the fish goes

toward to. The fish exhibiting the follow behavior will go toward other fish where food concentration is better. Let X_i be the current position of the artificial fish, and $y_{max} = max\{y(X_j)|X_j \epsilon s\}$. The behavior can be expressed as:

$$Follow(X_i^{t+1}) = \begin{cases} X_i^{(t)} + \frac{X_{max} - X_i^{(t)}}{||X_{max} - X_i^{(t)}||}.step.rand() & if \frac{y_{min}}{n_f} < \delta y_i \\ Prey(X_i^{(t+1)}) & else \end{cases}$$

2.7 Particle Swarm Optimization (PSO)

Particle swarm optimization was conceptualized by James Kennedy and Russel Eberhart[24]. The method which is stated in [24] "was discovered through simulation of simplified social model". It is based on artificial life and swarming theory and related to some evolutionary algorithms. Particle swarm optimization is an easy and simple concept which can be implemented with only a few lines of code. With its simplicity comes small memory requirements and faster acquisition of the optimal solution. The algorithm uses a modifying variable called velocity, which is updated and added to an individual. The individual will then undergo evaluation and will be carried on the next iteration if it has better fitness.

2.7.1 Velocity

Velocity as the name implies, is the speed of a particle towards a certain direction at a particular time interval. The velocity in PSO is added/subtracted for the population to be modified. The value of the velocity determines the convergence and the capacity of the PSO to find a good solution. The velocity is based on cognitive and social factors. The velocity is given by:

$$v_{id}^{k+1} = C[\omega \times v_{id}^{k} + c_1 rand_1 \times (pbest_{id} - x_{id}) + c_2 \times rand_2 \times (gbest_{gd} - x_{id})] \quad (2.38)$$

where C is the constriction factor, ω is the inertia weight parameter which controls the global and local exploration capabilities of the particle, c_1, c_2 are cognitive and social coefficients, respectively, and $rand_1$ and $rand_2$ are random numbers betwee 0 and 1.A

large cognitive coefficientc_1 means having the particle pull it to the local best position wheras a large social coefficient c_2 will pull the particle to the global position. A larger inertia weight factor is used during initial exploration its value is gradually reduced as the search proceeds. ω is given by:

$$\omega = (\omega_{max} - \omega_{min}) \times \frac{iter_{max} - iter}{iter_{max} + \omega_{min}} \tag{2.39}$$

where $iter_{max}$ is the maximum number of iterations. The constriction factor C is given by:

$$C = \frac{2}{|2 - \varphi - sqrt\varphi^2 - 4\varphi|} \tag{2.40}$$

where $4.1 \leq \varphi \leq 4.2$. As φ increases, the factor C decreases and convergence becomes slower because population diversity is reduced.

In updating the population, we modify an individual using:

$$x_{id}^{k+1} = x_{id} + v_{id}^{k+1} \tag{2.41}$$

where x_{id} is an element of an individual, X.

In evaluating the individuals, we introduce $pbest_i$ and $gbest$. $pbest_i$ is the best fitness of ith particle while $gbest$ is the global best fitness.

2.7.2 Cognitive Factor

The cognitive factor is the ability of the swarm to retain its original value. The cognitive factor balances the disruptive nature of the social factor. The higher or better the cognitive factor is, the better the local searching capability of a particle.

2.7.3 Social Factor

The social factor is the ability of a particle to follow other particle in search for a good solution. The social factor is based on the fittest particle. The better a global particle is, the more particles it will attract.

2.7.4 Time Varying Inertia Weight (TVIW)

Also called linearly varying inertia weight, the inertia weight ω is a variable that modifies the velocity in such manner that it accelarate the velocity from fast to slow. The modification is based on the assumption that particles should start with great velocity for faster convergence and slow down while the iteration nears to the max iteration. The slowing down of the velocity gives the velocity to have a good local searching capability since the step or velocity a particle make becomes smaller and smaller.

2.7.5 Time Varying Acceleration Coeffcients (TVAC)

A study of Eberhart and Shi found that TVIW is not very effective in solving dynamic and complicated systems. A new parameter that will replace TVIW is proposed. TVAC is a modification of TVIW and focuses more on the social factor of a particle to promote a good global searching capability.

2.8 Self-Organizing Hierarchical Particle Swarm Optimization with Time Varying Acceleration Coefficients

The self-organizing hierarchical particle swarm optimization was first presented in [6] and was applied in a related scheduling problem in [28].

SOHPSO prevents particles's velocity to stagnate. When the velocity reaches zero, it is reinitialized based on the maximum velocity set by the decision maker. The velocity is given by:

$$\begin{aligned} v_{id}^{k+1} = &((c_{if} - c_{1i})\frac{iter}{iter_{max}} + c_{1i}) \times rand_1 \times (pbest_{id} - x_{id}) \\ &+((c_{2f} - c_{2i})\frac{iter}{iter_{max}} + c_{2i}) \times rand_2 \times (gbest_{gd} - x_{id}) \end{aligned} \tag{2.42}$$

if $v_id = 0$ and $rand_3 < 0.5$ then $v_{id} = rand_4 \times V_{dmax}$
else $v_{id} = -rand_5 \times V_{dmax}$

where c_{1i}, c_{1f}, c_{2i}, and c_{2f} are initial and final values of cognitive and social acceleration factors, respectively. The initialization of velocity is given by:

$$v_i = U(-|L_{upper} - L_{lower}|, |L_{upper} - L_{lower}|) \tag{2.43}$$

where L is the limit. The initialization of the velocity is based on the upper and lower limits with respect to the equivalent variable of the velocity.

Chapter 3

Methodology

The mathematical model of thermal and hydro plant in the Review of Related Literature 2 and the objective function (2.6) are used in solving the SHS problem. Two test systems are introduced from [43, 19] and will be the test systems that will be optimized. The first test system is composed of 4 hydro plant and 1 thermal plant, with two cases. The first case for the first test system does not take in account the valve-point effects and transmission losses. The second case however applies the concept of valve-point efects. The second test system consists of 4 hydro plants and three thermal plants. The first case for the second test system applies the concept of the valve-point effects without considering transmission losses. In the second case, valve-point effects and transmission losses are considered.

The unit used for the input of the thermal plants is the cost of the fuel while the output is measured in megawatts(MW). For the hydro plant, the water discharge rate is measured in $\times 10^4 meters^3$.

3.1 Parameter Constants

Every individual and their chromosome is subjected to constraints handling. The following are the constraints given in [19]

3.1.1 Test System 1

$$P_s^{min} = 500 \quad P_s^{max} = 2500 \tag{3.1}$$

$$Q_1^{min} = 5 \quad Q_2^{min} = 6 \quad Q_3^{min} = 10 \quad Q_4^{min} = 13$$
$$Q_1^{max} = 15 \quad Q_2^{max} = 15 \quad Q_3^{max} = 30 \quad Q_4^{max} = 25$$

$$V_1^{min} = 80 \quad V_2^{min} = 60 \quad V_3^{min} = 100 \quad V_4^{min} = 170$$
$$V_1^{max} = 150 \quad V_2^{max} = 120 \quad V_3^{max} = 240 \quad V_4^{max} = 160$$

$$V_1^{begin} = 100 \quad V_2^{begin} = 80 \quad V_3^{begin} = 170 \quad V_4^{begin} = 120$$
$$V_1^{end} = 120 \quad V_2^{end} = 70 \quad V_3^{end} = 170 \quad V_4^{end} = 140$$

$$a_{si} = 5000 \quad b_{si} = 19.2 \quad c_{si} = 0.002 \quad d_{si} = 700 \quad e_{si} = 0.085$$

$$P_D[24] = [1370, 1390, 1360, 1290, 1290, 1410, 1650, 2000, 2240, 2320, 2230, 2310,$$
$$2230, 2200, 2130, 2070, 2130, 2140, 2240, 2280, 2240, 2120, 1850, 1590]$$

$$\tau_{kj}[4] = [2, 3, 4, 0]$$

plant	c_1	c_2	c_3	c_4	c_5	c_6
1	-0.0042	-0.42	0.030	0.90	10.0	-50
2	-0.0040	-0.030	0.015	1.14	9.5	-70
3	-0.0016	-0.30	0.014	0.55	5.5	-40
4	-0.0030	-0.31	0.027	1.44	14.0	-90

Table 3.1: Constants of thermal plants for test system

1

hour	reservoir 1	reservoir 2	reservoir 3	reservoir 4
1	10	8	8.1	2.8
2	9	8	8.2	2.4
3	8	9	4	1.6
4	7	9	2	0
5	6	8	3	0
6	7	7	4	0
7	8	6	3	0
8	9	7	2	0
9	10	8	1	0
10	11	9	1	0
11	12	9	1	0
12	10	8	2	0
13	11	8	4	0
14	12	9	3	0
15	11	9	3	0
16	10	8	2	0
17	9	7	2	0
18	8	6	2	0
19	7	7	1	0
20	6	8	1	0
21	7	9	2	0
22	8	9	2	0
23	9	8	1	0
24	10	8	0	0

Table 3.2: Water reservoir inflow for all test system

3.1.2 Test System 2

$$P_smin[3] = [20, 40, 50]$$

$$P_smax[3] = [176, 300, 500]$$

$$Q_1^{min} = 5 \quad Q_2^{min} = 6 \quad Q_3^{min} = 10 \quad Q_4^{min} = 13$$
$$Q_1^{max} = 15 \quad Q_2^{max} = 15 \quad Q_3^{max} = 30 \quad Q_4^{max} = 25$$

$$V_1^{min} = 80 \quad V_2^{min} = 60 \quad V_3^{min} = 100 \quad V_4^{min} = 170$$
$$V_1^{max} = 150 \quad V_2^{max} = 120 \quad V_3^{max} = 240 \quad V_4^{max} = 160$$

$$V_1^{begin} = 100 \quad V_2^{begin} = 80 \quad V_3^{begin} = 170 \quad V_4^{begin} = 120$$
$$V_1^{end} = 120 \quad V_2^{end} = 70 \quad V_3^{end} = 170 \quad V_4^{end} = 140$$

$$a_si[3] = [100, 120, 150]$$

$$b_si[3] = [2.45, 2.32, 2.10]$$

$$c_si[3] = [0.0012, 0.0010, 0.0015]$$

$$d_si[3] = [160, 180, 200]$$

$$e_si[3] = [0.038, 0.037, 0.035]$$

$$P_D[24] = [750, 780, 700, 650, 670, 800, 950, 1010, 1090, 1080, 1100, 1150, 1110, 1030, 1010,$$

$$1060, 1050, 1120, 1070, 1050, 910, 860, 850, 800]$$

Constants for hydro plant is the same in test system 1.

3.2 Data Representation

An individual in a population is represented by X_i for the ith individual given by:

$$X_n = \begin{bmatrix} Q_h(1,1) & Q_h(2,1) & \dots & Q_h(N_h,1) & P_s(1,1) & P_s(2,1) & \dots & P_s(N_s,1) \\ Q_h(1,2) & Q_h(2,2) & \dots & Q_h(N_h,2) & P_s(1,2) & P_s(2,2) & \dots & P_s(N_s,2) \\ \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \dots & \vdots \\ Q_h(1,T) & Q_h(2,T) & \dots & Q_h(N_h,T) & P_s(1,1) & P_s(2,1) & \dots & P_s(N_h,T) \end{bmatrix} \quad (3.2)$$

While the whole population is denoted by:

$$X = [X_1, X_2 \ldots, X_N] \tag{3.3}$$

where N is the number of individuals in a population. The fitness value for each individuals is given by:

$$F = [F_1, F_2, \ldots, F_N] \tag{3.4}$$

3.3 Initialization of Population

The initial population will be generated using

$$\begin{aligned} Q_h(j,t) &= Q_h(j)^{min} + U(0,1) \times \left(Q_h(j)^{max} - Q_h(j)^{min}\right) \\ P_s(i,t) &= P_s(i)^{min} + U(0,1) \times \left(P_s(i)^{max} - P_s(i)^{min}\right) \end{aligned} \tag{3.5}$$

where U is a randomly generated number from 0 to 1. $Q_h(j)^{max}$ and $Q_h(j)^{min}$ are the upper and lower limits of the water discharge rate$P_s(i)^{min}$ and $P_s(i)^{min}$ are the upper and lower limits of the power generation of the ith thermal plant. As an example for $Q_h(1,1)$, we generate a random number $U = 0.1286$, where $Q_h(1)^{min} = 5$ and $Q_h(1)^{max} = 15$. We plug in U in:

$$\begin{aligned} Q_h(1,1) &= Q_h(1)^{min} + 0.129 \times \left(Q_h(1)^{max} - Q_h(1)^{min}\right) \\ Q_h(1,1) &= 5 + 0.129 \times (15 - 5) \\ Q_h(1,1) &= 6.286 \end{aligned} \tag{3.6}$$

and for $P_s(1,1)$, we generate $U = 0.057$, where $P_s(i)^{min} = 500$ and $P_s(i)^{max} = 2500$. We plug in the values in:

$$\begin{aligned} P_s(1,1) &= P_s(i)^{min} + U(0,1) \times \left(P_s(i)^{max} - P_s(i)^{min}\right) \\ P_s(1,1) &= P_s(i)^{min} + 0.057 \times \left(P_s(i)^{max} - P_s(i)^{min}\right) \\ P_s(1,1) &= 500 + 0.057 \times (2500 - 500) \\ P_s(1,1) &= 614.473 \end{aligned} \tag{3.7}$$

The process is done for all N_h hydro plants and N_s thermal plants in all time interval up to T. If we have $N_h = 4$ and $N_s = 1$, the whole individual will be:

$$X_1 = \begin{bmatrix} & & & & \overbrace{\quad}^{P_s(i,t)} \\ 6.286 & 10.888 & 17.728 & 19.139 & 614.473 \\ 7.025 & 6.495 & 24.228 & 13.916 & 1408.858 \\ 9.248 & 8.603 & 19.135 & 19.220 & 508.199 \\ 10.853 & 13.106 & 20.974 & 15.026 & 1306.477 \\ 5.048 & 12.754 & 10.884 & 15.423 & 2140.580 \\ 10.289 & 12.332 & 23.131 & 15.012 & 1802.334 \\ 5.801 & 12.407 & 21.384 & 16.137 & 1298.278 \\ 11.635 & 6.186 & 26.206 & 16.807 & 1144.574 \\ 10.374 & 8.144 & 16.826 & 14.621 & 1117.317 \\ 5.792 & 10.457 & 27.300 & 15.581 & 1603.367 \\ 9.368 & 13.049 & 13.471 & 16.530 & 1888.533 \\ 13.561 & 13.268 & 24.568 & 20.491 & 2116.780 \\ 8.961 & 7.472 & 14.896 & 18.748 & 2169.661 \\ 10.806 & 11.287 & 10.917 & 19.594 & 732.386 \\ 7.481 & 10.335 & 23.684 & 21.219 & 2366.600 \\ 8.263 & 13.933 & 15.528 & 19.541 & 645.905 \\ 13.098 & 8.009 & 19.400 & 13.771 & 1479.827 \\ 12.416 & 13.345 & 21.114 & 16.716 & 2099.820 \\ 7.450 & 13.526 & 15.376 & 14.729 & 949.400 \\ 13.743 & 14.092 & 23.235 & 24.775 & 2124.337 \\ 14.890 & 10.207 & 28.781 & 18.824 & 540.201 \\ 14.325 & 8.338 & 21.022 & 13.481 & 1462.222 \\ 9.930 & 7.664 & 16.247 & 16.139 & 2355.188 \\ 12.175 & 13.470 & 23.892 & 22.326 & 995.791 \end{bmatrix} \tag{3.8}$$

which takes the form of the matrix 3.2. The last column are the power generation of thermal plants $P_s(i,t)$ for all time interval t up to T.

3.4 Evaluation

The objective function is given by:

$$F_{totalcost} = \sum_{t=1}^{T} \sum_{i=1}^{N_s} \left[\left(a_{si} + b_{si} P_s(i,t) + c_{si} (P_s(i,t))^2 \right) + |d_{si} \sin(e_{si}(P_s(i)^{min} - P_s(i,t)))| \right] \tag{3.9}$$

based on the modeling done in 2.1.7. If we use the X_1 3.8, we get all the thermal plant power generation for all time interval and use the objective function. When the process

is expanded, it will look like:

$$\begin{aligned}
F_1 = \sum_{t=1}^{T}\sum_{i=1}^{N_s} &\left[\left(a_{si} + b_{si}\cdot P_s(i,t) + c_{si}\cdot (P_s(i,t))^2\right) + |d_{si}\cdot \sin(e_{si}\cdot (P_s(i)^{min} - P_s(i,t)))|\right] \\
&= \left[\left(5000 + 19.2\cdot P_s(i,t) + 0.002\cdot (P_s(i,t))^2\right) + |700\cdot \sin(0.085\cdot (500 - P_s(i,t)))|\right] \\
&= \left[\left(5000 + 19.2\cdot 614.473 + 0.002\cdot (614.473)^2\right) + |700\cdot \sin(0.085\cdot (500 - 614.473))|\right] + \\
&\left[\left(5000 + 19.2\cdot 1408.858 + 0.002\cdot (1408.858)^2\right) + |700\cdot \sin(0.085\cdot (500 - 1408.858))|\right] + \\
&\vdots \\
&\left[\left(5000 + 19.2\cdot 995.791 + 0.002\cdot (995.791)^2\right) + |700\cdot \sin(0.085\cdot (500 - 995.791))|\right] + \\
&F_1 = 908052.616
\end{aligned} \tag{3.10}$$

By inserting the values of rightmost elements of the X_1 matrix, which are the thermal power generation, $P_s(i,t)$, disregarding the valve-point effect, we produce the fitness value which is 44926.8. The fitness value will then be stored to another matrix that holds every fitness value of every individual in the population. The fitness matrix, F_{n_f} will have its first element, \$44926.8 in $F_1 = \$44926.8$. F is defined as:

$$F = [44926.8, \ldots, F_N]$$

After evaluating every individual, they will be sorted from the minimum to maximum. The first individual is the most fit individual in the population.

3.5 Constraints Handling

The constraints handling method handles the elements that violates the system constraints. The method is also responsible for the distribution of the water discharge load and the power generation balance while satisfying various system constraints.

3.5.1 Hydro Plants

In the constraints handling method of the hydro plants, we first get $\Delta V_h(j)$ which is the volume violation of the jth hydro plant with respect to the final reservoir volume. $\Delta V_h(j)$ is given by:

$$\Delta V_h(j) = V_h(j,T) - V_h(j)^{end} \tag{3.11}$$

The $\Delta V_h(j)$ will be distributed to all the water reservoir with respect to time by controlling the water discharge rate $Q_h(j,t)$. The distribution of $\Delta V_h(j)$ is done to meet the final water reservoir volume imposed by the decision maker. For example, if we have $\Delta V_h(1) = 15$ and we have X_1 as an individual where X_1 is :

$$X_1 = \begin{bmatrix} \overbrace{6.286}^{Q_h(1,t)} & \overbrace{10.888}^{Q_h(2,t)} & \overbrace{17.728}^{Q_h(3,t)} & \overbrace{19.139}^{Q_h(4,t)} & \overbrace{614.473}^{P_s(i,t)} \\ 7.025 & 6.495 & 24.228 & 13.916 & 1408.858 \\ 9.248 & 8.603 & 19.135 & 19.220 & 508.199 \\ 10.853 & 13.106 & 20.974 & 15.026 & 1306.477 \\ 5.048 & 12.754 & 10.884 & 15.423 & 2140.580 \\ 10.289 & 12.332 & 23.131 & 15.012 & 1802.334 \\ 5.801 & 12.407 & 21.384 & 16.137 & 1298.278 \\ 11.635 & 6.186 & 26.206 & 16.807 & 1144.574 \\ 10.374 & 8.144 & 16.826 & 14.621 & 1117.317 \\ 5.792 & 10.457 & 27.300 & 15.581 & 1603.367 \\ 9.368 & 13.049 & 13.471 & 16.530 & 1888.533 \\ 13.561 & 13.268 & 24.568 & 20.491 & 2116.780 \\ 8.961 & 7.472 & 14.896 & 18.748 & 2169.661 \\ 10.806 & 11.287 & 10.917 & 19.594 & 732.386 \\ 7.481 & 10.335 & 23.684 & 21.219 & 2366.600 \\ 8.263 & 13.933 & 15.528 & 19.541 & 645.905 \\ 13.098 & 8.009 & 19.400 & 13.771 & 1479.827 \\ 12.416 & 13.345 & 21.114 & 16.716 & 2099.820 \\ 7.450 & 13.526 & 15.376 & 14.729 & 949.400 \\ 13.743 & 14.092 & 23.235 & 24.775 & 2124.337 \\ 14.890 & 10.207 & 28.781 & 18.824 & 540.201 \\ 14.325 & 8.338 & 21.022 & 13.481 & 1462.222 \\ 9.930 & 7.664 & 16.247 & 16.139 & 2355.188 \\ 12.175 & 13.470 & 23.892 & 22.326 & 995.791 \end{bmatrix} \tag{3.12}$$

We calculate for $averageV_h(1) = \Delta V_h(1)/T$, where $T = 24$.

$$\begin{aligned} averageV_h(j) &= \Delta V_h(j)/T \\ averageV_h(j) &= 48/24 \\ averageV_h(j) &= 0.625 \end{aligned} \tag{3.13}$$

Applying the changes to X_1

$$X_1 = \begin{bmatrix} \overbrace{6.286}^{Q_h(1,t)} & 10.888 & 17.728 & 19.139 & 614.473 \\ 7.65 & 6.495 & 24.228 & 13.916 & 1408.858 \\ 9.873 & 8.603 & 19.135 & 19.220 & 508.199 \\ 11.478 & 13.106 & 20.974 & 15.026 & 1306.477 \\ 5.673 & 12.754 & 10.884 & 15.423 & 2140.580 \\ 10.914 & 12.332 & 23.131 & 15.012 & 1802.334 \\ 6.425 & 12.407 & 21.384 & 16.137 & 1298.278 \\ 12.26 & 6.186 & 26.206 & 16.807 & 1144.574 \\ 10.999 & 8.144 & 16.826 & 14.621 & 1117.317 \\ 6.417 & 10.457 & 27.300 & 15.581 & 1603.367 \\ 9.993 & 13.049 & 13.471 & 16.530 & 1888.533 \\ 14.186 & 13.268 & 24.568 & 20.491 & 2116.780 \\ 9.586 & 7.472 & 14.896 & 18.748 & 2169.661 \\ 11.431 & 11.287 & 10.917 & 19.594 & 732.386 \\ 8.106 & 10.335 & 23.684 & 21.219 & 2366.600 \\ 8.888 & 13.933 & 15.528 & 19.541 & 645.905 \\ 13.723 & 8.009 & 19.400 & 13.771 & 1479.827 \\ 13.041 & 13.345 & 21.114 & 16.716 & 2099.820 \\ 8.075 & 13.526 & 15.376 & 14.729 & 949.400 \\ 14.368 & 14.092 & 23.235 & 24.775 & 2124.337 \\ 15.515 & 10.207 & 28.781 & 18.824 & 540.201 \\ 14.95 & 8.338 & 21.022 & 13.481 & 1462.222 \\ 10.555 & 7.664 & 16.247 & 16.139 & 2355.188 \\ 12.8 & 13.470 & 23.892 & 22.326 & 995.791 \end{bmatrix} \tag{3.14}$$

The process is done for $iteration_{course}$ times to every column which are the jth hydro plant or the ith thermal plant.

After the process of course adjusting the $\Delta V_h(j)$, there will still be a final water reservoir volume violation that can be found since the previous process was only done for $iteration_{course}$ times. To handle the remaining $\Delta V_h(j)$, we find an r with the maximum adjustable discharge rate $Q_h(j,t)$. For example, if $\Delta V_h(j) = 0.25$ then $r = 22$. The new population will be:

$$X_1 = \begin{bmatrix} \overbrace{6.286}^{Q_h(1,t)} & 10.888 & 17.728 & 19.139 & 614.473 \\ 7.65 & 6.495 & 24.228 & 13.916 & 1408.858 \\ 9.873 & 8.603 & 19.135 & 19.220 & 508.199 \\ 11.478 & 13.106 & 20.974 & 15.026 & 1306.477 \\ 5.673 & 12.754 & 10.884 & 15.423 & 2140.580 \\ 10.914 & 12.332 & 23.131 & 15.012 & 1802.334 \\ 6.425 & 12.407 & 21.384 & 16.137 & 1298.278 \\ 12.26 & 6.186 & 26.206 & 16.807 & 1144.574 \\ 10.999 & 8.144 & 16.826 & 14.621 & 1117.317 \\ 6.417 & 10.457 & 27.300 & 15.581 & 1603.367 \\ 9.993 & 13.049 & 13.471 & 16.530 & 1888.533 \\ 14.186 & 13.268 & 24.568 & 20.491 & 2116.780 \\ 9.586 & 7.472 & 14.896 & 18.748 & 2169.661 \\ 11.431 & 11.287 & 10.917 & 19.594 & 732.386 \\ 8.106 & 10.335 & 23.684 & 21.219 & 2366.600 \\ 8.888 & 13.933 & 15.528 & 19.541 & 645.905 \\ 13.723 & 8.009 & 19.400 & 13.771 & 1479.827 \\ 13.041 & 13.345 & 21.114 & 16.716 & 2099.820 \\ 8.075 & 13.526 & 15.376 & 14.729 & 949.400 \\ 14.368 & 14.092 & 23.235 & 24.775 & 2124.337 \\ 15.515 & 10.207 & 28.781 & 18.824 & 540.201 \\ 15.2 & 8.338 & 21.022 & 13.481 & 1462.222 \\ 10.555 & 7.664 & 16.247 & 16.139 & 2355.188 \\ 12.8 & 13.470 & 23.892 & 22.326 & 995.791 \end{bmatrix} \tag{3.15}$$

The constraints handling method for hydro plants also handles values that go out of their bounds. All of the elements will be checked if they violate any of their constraints. In the individual X_1, we can see that $Q_h(1, r)$ violates its limits, so its value will be changed to the maximum water discharge rate

$$Q_h(1, r) = Q_h(1)^{max} = 15$$

for the 1_{st} hydro plant.

The whole process is shown as pseudocode 2.2.1.

3.5.2 Thermal Plants

The same line of process is used for the constraints handling method for the thermal plants. Given a power violation $\Delta P(t)$ which is given by:

$$\Delta P(t) = P_D(t) + P_L(t) - \sum_{i=1}^{N_s} P_s(i, t) - \sum_{j=1}^{N_h} P_h(j, t) \tag{3.16}$$

where $P_D(t)$ is power demand and $P_L(t)$ is the power transmission loss. For example,

if we have $\Delta P(t) = 500$, we define

$$averageP(t) = \Delta P(t)/N_s$$

where N_s is the number of thermal plants. Since in X_1 there is only one thermal plant, the $averageP(t)$ will be distributed to the only thermal plant. The process will be iterated for $iteration_{course}$ times.

After the course adjustments we select r_{th} thermal plant with the maximum adjustable generation for the fine adjustments. $\Delta P(t)$ will then be loaded to the r_{th} thermal plant shown by:

$$P_s(r,t) = P_s(r,t) + \Delta P(t) \tag{3.17}$$

The fine adjustment will be repeated $iteration_{fine}$ times.

The whole process is shown as a pseudocode in 2.1.1.

3.6 Real-Coded Genetic Algorithm

Real-coded genetic algorithm is a stochastic search heuristics based on the natural phenomenon of natural selection and evolution that uses real values. A classical genetic algorithm consists of steps or genetic operators that vary and changes the population in the hopes of acquiring the optimal solution. The genetic operators are called selection, crossover and mutation. In the selection, individuals are weighed and their fitness value calculated. They then select the most promising individuals and use them to further improve the solution.

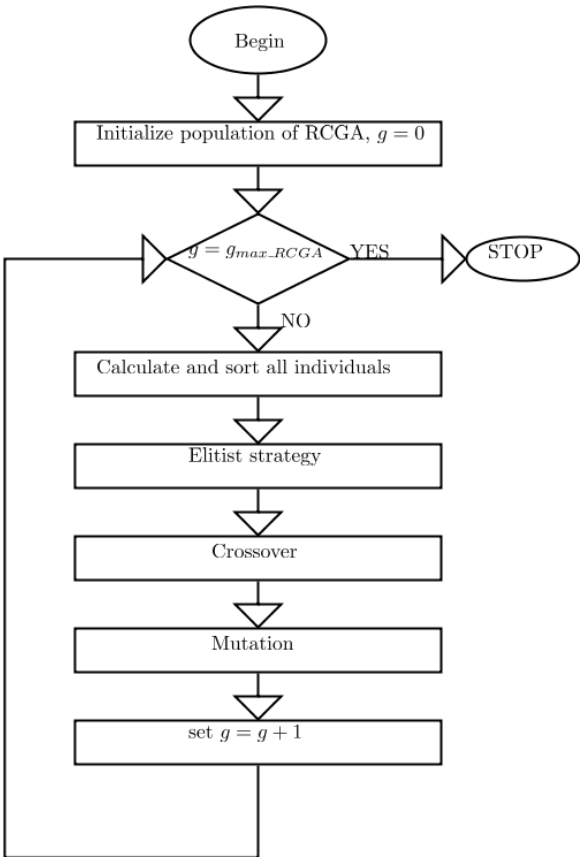


Figure 3.1: Flowchart of real-coded genetic algorithm

3.6.1 Selection

Elitist Strategy is an operator where the best parents are carried over to the next generation without alteration of their genes. In RCGA, we get P_e, the elitist population of individuals from P_N, the whole population of the algorithm. P_e is then removed from the whole population and will be preserved to be carried over to the next iteration while the remaining population will undergo crossover and mutation. Elitist Strategy is done to balance the disruptive process of the crossover and mutation operator instead of having the whole population undergo crossover and mutation. In comparison to having the whole population undergo crossover and mutation and applying elitist strategy, elitist is more ideal since preserving the best individuals is likely to yield the best population. From the process, if the global optimal solution is generated, it will be preserved instead

of undergoing the crossover and mutation operator which can alter the global optimal solution. As an example, if we have a total of $N_p = 100$ number of population and $P_e = 30$, the N_p population will be sorted based on their fitness value where the first value is the best fitness and preserve the frist 30 individuals in the population.

3.6.2 Crossover

Simulated binary crossover is used as the type of crossover. Individuals P_c will be taken from $(N_p - P_e)$. For the individual X_1 defined earlier, we introduce another individual, X_2 which will be used in the crossover.

$$X_2 = \begin{bmatrix} 12.4 & 6.3 & \dots & 138.093 \\ 7.90411 & 7.89401 & \dots & 142.592 \\ \vdots & \vdots & \dots & \vdots \\ 7.90613 & 8.51025 & \dots & 132.609 \end{bmatrix}$$

Two parent individuals are needed for the production of two children. We first get one chromosome from X_1, which is $X_1(1,1) = 11.4457$ and $x_2(1,1) = 12.4$. We then generate a uniform random number u between 0 and 1. Let $u = 0.5$. We then calculate

$$\beta = 1 + \frac{2}{x_2^{(1,1)} - x_1^{1,1}} min[(x_1^{(1,1)} - x_l^{1,1}), (x_u^{1,1} - x_2^{(1,1)})]$$

$$\beta = 1 + \frac{2}{12.4 - 11.4457} min[(11.4457 - 5), (15 - 12.4)]$$

$$\beta = 6.4490$$

After calculating for β, which is the spread factor given by [17], we then solve for α, where

$$\alpha = 2 - \beta^{-\eta_c + 1}$$

$$\alpha = 1.976$$

where η_c is the distribution index with nonnegative value. A small value of η_c gives children solutions that are far away from the parents and a large value gives near parent solutions. We use the value of $\eta_c = 1$ to minimize the disruptive nature of crossover

and ensuring that the crossover still works. After calculating for α, we calculate the polynomial probability distribution γ:

$$\gamma = \begin{cases} (\alpha u)^{\frac{1}{\eta_c+1}} & if u \leq \frac{1}{\alpha} \\ (\frac{1}{2-\alpha u})^{\frac{1}{\eta_c+1}} & otherwise \end{cases}$$

The first case is used since $u = 0.5 \leq 0.506$.

$$\gamma = 0.5030$$

After calculating SBX parameters β, α, and γ, we create the children individuals given by the equation:

$$y_1^{(1,1)} = 0.5[(x_1^{(1,1)} + x_2^{(1,1)}) - \gamma|x_2^{(1,1)} - x_1^{(1,1)}|]$$

$$y_2^{(1,1)} = 0.5[(x_1^{(1,1)} + x_2^{(1,1)}) + \gamma|x_2^{(1,1)} - x_1^{(1,1)}|]$$

$$y_i^{(1)} = 0.5[(11.4457 + 12.4) - \gamma|12.4 - 11.4457|]$$

$$y_i^{(1)} = 0.5[(11.4457 + 12.4) + \gamma|12.4 - 11.4457|]$$

$$y_i^{(1)} = 11.6828$$

$$y_i^{(2)} = 12.1629$$

where $y_i^{(1)}$ and $y_i^{(2)}$ are the new chromosomes for for $Y_i^{(1)}$ and $Y_i^{(2)}$ children individuals.

$$Y_1 = \begin{bmatrix} 11.6828 & \dots \\ \vdots & \dots \end{bmatrix}$$

$$Y_2 = \begin{bmatrix} 12.1629 & \dots \\ \vdots & \dots \end{bmatrix}$$

The SBX operator will be applied to all members/chromosome of the parent chromosome to produce children. The children will be carried over to the next generation while the parents will be discarded.

3.6.3 Mutation

Mutation operator mutates one individual at a time. We get P_m from $(N_p - P_e)$. We first generate r, which is a uniformly distributed random number in $[0, 1]$. We use X_1 for the example. We first generate τ to calculate:

$$X_i^{'} = \begin{cases} x_i + \Delta(t, x_i^u - x_i) & (\tau = 0) \\ x_i - \Delta(t, x_i^u - x_i) & (\tau = 1) \end{cases}$$

where we generate $r = 0.5$, which is a uniformly distributed random number in $[0, 1]$. $\Delta(t, y)$ is

$$\Delta(t, y) = y(1 - r^{(1-\frac{t}{t_{max}})^b})$$

where $b = 5$ [35]. Let $\tau = 1$ and solve the first element of X_1

$$x_1^{(1,1)} = 11.4457$$

$$x_1^{\prime(1,1)} = x_i - \Delta(1, 15 - 11.4457)$$

where

$$\Delta(1, 3.5543) = 3.5543(1 - 0.5^{(1-\frac{1}{24})^5})$$

$$\Delta(1, 3.5543) = 0.80832$$

$$x_1^{\prime(1,1)} = 11.4457 - 0.80832$$

$$x_1^{\prime(1,1)} = 10.63738$$

the mutated chromosome will be stored in $X_1^{(1,1)}$,

$$X_1 = \begin{bmatrix} 10.63738 & \dots \\ \vdots & \dots \end{bmatrix}$$

All chromosome of the mutated individual will be subjected to the same process of mutation. The mutated individual will then be carried over to the next generation.

3.7 Artificial Fish Swarm Algorithm

Artificial fish swarm algorithm is based on the three main behaviors of a school of fish. These behaviors are prey, swarm and follow behavior.

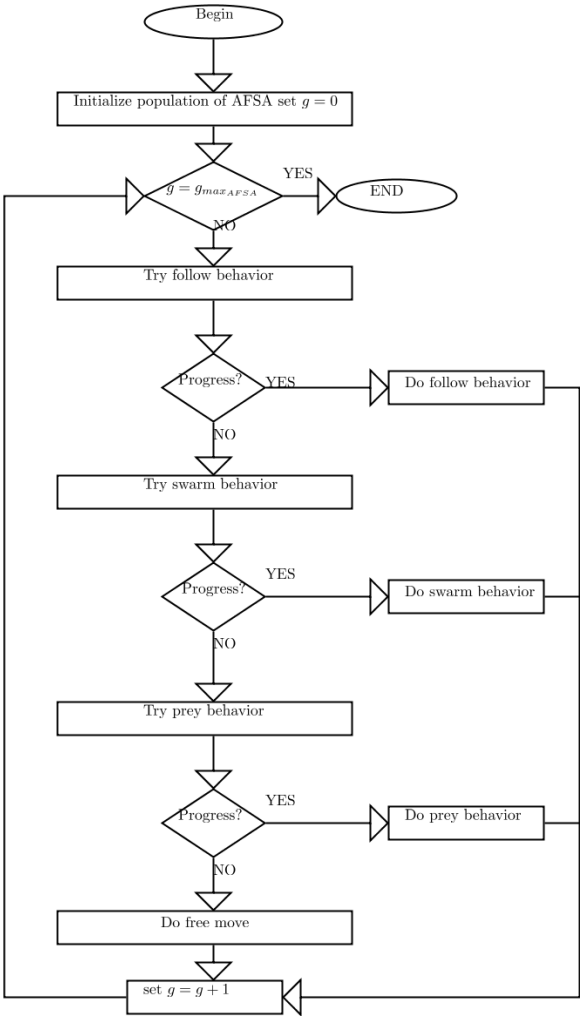


Figure 3.2: Flowchart of artificial fish swarm algorithm

The flow of AFSA is give by Figure 3.2.

3.7.1 Swarm Behavior

We use an individual X_1 for example. The Y variable is equal to F, which is the fitness values of every individual. The individual fish X_1 will search for all the nearest fishes

within its visual and will determine the center of the swarm. The chromosomes of all the nearest fishes, which are X_j to X_{n_f} will be subjected to

$$\sum_{j=1}^{n_f} \frac{x_j}{n_f}$$

In implementing AFSA, we follow Figure 3.2. We use an individual X_s for example. X_1 is copied to X_s, which is an individual in a population. The y variable is equal to F, which is the fitness values of every individual. The individual fish X_s will search for all the nearest fishes within its visual and will determine the center of the swarm. The chromosomes of all the nearest fishes, which are X_j to X_{n_f} will be subjected to

$$\sum_{j=1}^{n_f} \frac{x_j}{n_f}$$

where n_f is the number of fishes within the visual distance of the X_1 fish. The equation will be equal to X_c, which is the average position, or the center of the swarm. For example, if we have three individuals, X_2, X_3, and X_4, within a fish's visual, we get the average/center of the swarm and will be equal to X_c

$$X_s = \begin{bmatrix} 11.4457 & 6.3 & \dots \\ 7.90411 & 7.89401 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

$$X_2 = \begin{bmatrix} 10 & 9 & \dots \\ 9 & 10 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

$$X_3 = \begin{bmatrix} 9 & 9 & \dots \\ 10 & 10 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

$$X_4 = \begin{bmatrix} 8 & 9 & \dots \\ 8 & 7 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

then

$$X_c = \begin{bmatrix} 9 & 9 & \dots \\ 9 & 9 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

We then implement the swarm behavior's main method

$$Swarm(X_s) = \begin{cases} X_s + \frac{X_c - X_s}{||X_c - X_s||} \cdot step \cdot rand() & if \frac{y_c}{n_f} < \delta y_i \\ Prey(X_s) & else \end{cases}$$

where $||X_c - X_s||$ is the euclidian distance between X_c and $X_i^{(t)}$, y_c is the summation of all fitness values of the fishes within the visual of the X_1 fish, and δ is the crowd factor where

$$||X_c - X_1|| = \sqrt{(X_c(1,1) - X_1(1,1))^2 + (X_c(1,2) - X_1(1,2))^2 + \dots + (X_c(2,1) - X_1(2,1))^2 + (X_c(2,2) - X_1(2,2))^2 + \dots + (X_c(t, j+i) - X_1(t, j+i))^2} \tag{3.18}$$

$$||X_c - X_1|| = \sqrt{(9 - 11.4457)^2 + (9 - 6.3)^2 + \dots + (9 - 7.90411)^2 + (9 - 7.89401)^2 + \dots + (X_c(t, j+i) - X_1(t, j+i))^2} \tag{3.19}$$

The center of the swarm X_c is the average of all fishes within the visual of the fish. If the average fitness of the fishes within the visual of the X_s fish

$$\frac{y_c}{n_f}$$

where

$$y_c = F_2 + F_3 + F_4$$

is less than δF_s then the swarm behavior will be implemented. The process will affect all the elements of the X_s fish. If $||X_c - X_s|| = 0.5$ and $y_c = 874238.213$, and $\delta y_i = 908052.616$, $rand() = 0.5$, $step = 1$ then

$$X_s = \begin{bmatrix} 11.4457 & 6.3 & \dots \\ 7.90411 & 7.89401 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

will be

$$X_s = \begin{bmatrix} 10.22285 & 7.65 & \dots \\ 8.802055 & 8.447005 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

The newly generated X_s with its new F_s calculated from the objective/fitness function will be temporarily stored.

3.7.2 Follow Behavior

A fish will follow another fish with an area of more abundant food. We first select the best fish within the visual of the current fish, X_i, which is given by X_{min}, where $F_{min} = min\{F_j | j \epsilon s\}$, where s is the set of indices of fishes within the visual of the current fish. We then implement the cases of the follow behavior:

$$Follow(X_f) = \begin{cases} X_f + \frac{X_{min} - X_f}{||X_{min} - X_f||} \cdot step \cdot rand() & if F_{min} < \delta F_i \\ Prey(X_f) & else \end{cases}$$

As an example, given X_1, X_1 will be copied to X_f with a fitness value of F_f. The X_f follows the structure of $X_1 3.8$

$$X_f = \begin{bmatrix} 6.286 & 10.888 & \dots \\ 7.025 & 6.495 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.20}$$

Let

$$F_f = 908052.616$$

If given X_{min}:

$$X_{min} = \begin{bmatrix} 9 & 9 & \dots \\ 9 & 9 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.21}$$

with $F_{min} = 674238.213$ and $\delta = 0.8$. then

$$F_{min} < \delta y_i$$

$$674238.213 < 726442.093$$

holds true. The follow method will then be implemented. Given the Euclidian distance $||X_{min} - X_f|| = 1$, $rand() = 0.5$, $step = 1$

$$X_f = \begin{bmatrix} 6.286 & 10.888 & \dots \\ 7.025 & 6.495 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.22}$$

will be

$$X_f = \begin{bmatrix} 7.643 & 9.944 & \dots \\ 8.0125 & 7.7475 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.23}$$

The newly modified X_f will be temporarily restored with its new F_f calculated using the fitness/objective function.

3.7.3 Prey behavior

The prey behavior is implemented when neither swarm nor follow behavior acquired the better solution. X_1 3.8 will be copied to X_p along with its fitness value $F_p = F_1$. A randomy selected X_j area within the visual of the X_p fish is selected. The generation of X_j is based on the step size multiplied by a a randomly generated $rand()$ number. If

$$X_p = \begin{bmatrix} 6.286 & 10.888 & \dots \\ 7.025 & 6.495 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.24}$$

and $rand() = 0.5$ and $(visualdistance) = 1$, then

$$X_j = \begin{bmatrix} 6.786 & 11.388 & \dots \\ 7.525 & 6.995 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.25}$$

Prey behavior is expressed as

$$Prey(X_p) = \begin{cases} X_p + \frac{X_j - X_p}{||X_j - X_p||} \cdot step \cdot rand() & if F_j < F_p \\ else & X_p + step \cdot rand() \end{cases}$$

if $step = 1$, $rand() = 0.5$, $F_p = 908052.616$ and $F_j = 874238.213$, then the $F_j < F_p$ holds true. The first case of prey behavior will be implemented: If

$$X_p = \begin{bmatrix} 6.286 & 10.888 & \dots \\ 7.025 & 6.495 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.26}$$

then the new X_p based on X_j and the first case of prey behavior will be

$$X_p = \begin{bmatrix} 6.536 & 11.138 & \dots \\ 7.275 & 6.745 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.27}$$

The newly generated X_p and its new F_p computed from the fitness function will be stored temporarily. If the prey function is inside the swarm behavior, $X_s = X_p$ and $F_s = F_p$. If the prey function is inside the follow behavior, $X_f = X_p$ and $F_f = F_p$

Given X_s and X_f, the fitness value F_s and F_f will be compared. The best fitness will be chosen and will be the new X_1 artificial fish in the population. The process of artificial fish swarm algorithm will be iterated over all the individuals in the population until a certain number of iteration is met. After every process of AFSA, the whole population will undergo constraints handling method 3.5.

3.8 Particle Swarm Optimization with time-varying inertia weight

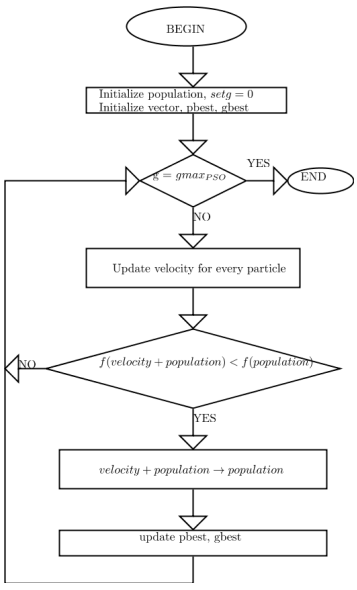


Figure 3.3: Flowchart of particle swarm optimization

A population of vectors is given by $V = [v_1, v_2, \ldots, v_N]$ where v_i is the respective velocity of the ith individual in the population of solutions.

3.8.1 Initialization of Velocity

The algorithm uses a modifying variable called velocity, which is updated and added to an individual. The individual will then undergo evaluation and will be carried on the next iteration if it has better fitness. Given an individual X_i

$$X_1 = \begin{bmatrix} & & & & \overbrace{}^{P_s(i,t)} \\ 6.286 & 10.888 & 17.728 & 19.139 & 614.473 \\ 7.025 & 6.495 & 24.228 & 13.916 & 1408.858 \\ 9.248 & 8.603 & 19.135 & 19.220 & 508.199 \\ 10.853 & 13.106 & 20.974 & 15.026 & 1306.477 \\ 5.048 & 12.754 & 10.884 & 15.423 & 2140.580 \\ 10.289 & 12.332 & 23.131 & 15.012 & 1802.334 \\ 5.801 & 12.407 & 21.384 & 16.137 & 1298.278 \\ 11.635 & 6.186 & 26.206 & 16.807 & 1144.574 \\ 10.374 & 8.144 & 16.826 & 14.621 & 1117.317 \\ 5.792 & 10.457 & 27.300 & 15.581 & 1603.367 \\ 9.368 & 13.049 & 13.471 & 16.530 & 1888.533 \\ 13.561 & 13.268 & 24.568 & 20.491 & 2116.780 \\ 8.961 & 7.472 & 14.896 & 18.748 & 2169.661 \\ 10.806 & 11.287 & 10.917 & 19.594 & 732.386 \\ 7.481 & 10.335 & 23.684 & 21.219 & 2366.600 \\ 8.263 & 13.933 & 15.528 & 19.541 & 645.905 \\ 13.098 & 8.009 & 19.400 & 13.771 & 1479.827 \\ 12.416 & 13.345 & 21.114 & 16.716 & 2099.820 \\ 7.450 & 13.526 & 15.376 & 14.729 & 949.400 \\ 13.743 & 14.092 & 23.235 & 24.775 & 2124.337 \\ 14.890 & 10.207 & 28.781 & 18.824 & 540.201 \\ 14.325 & 8.338 & 21.022 & 13.481 & 1462.222 \\ 9.930 & 7.664 & 16.247 & 16.139 & 2355.188 \\ 12.175 & 13.470 & 23.892 & 22.326 & 995.791 \end{bmatrix} \tag{3.28}$$

The velocity have the same vector size as X_1. Every elements in an individual have their own velocity. The initialization of velocity is given by:

$$\begin{aligned} v_{(t,j)} &= U(-|Q_h^{max}[j] - Q_h^{min}[j]|, |Q_h^{max}[j] - Q_h^{min}[j]|) \\ v_{(t,i)} &= U(-|P_s^{max}[i] - P_s^{min}[i]|, |P_s^{max}[i] - P_s^{min}[i]|) \end{aligned} \tag{3.29}$$

where $Q_h^{max}[j]$ and $Q_h^{min}[j]$ are the maximum and minimum water discharge of the jth hydro plant. $P_s^{max}[i]$ and $P_s^{min}[i]$ are the maximum and minimum power generation

of the *ith* thermal plant. An example velocity vector is given by:

$$v_1 = \begin{bmatrix} 3.83873 & -0.30638 & -2.83883 & -0.120536 & 201.403 \\ 2.83554 & 0.872582 & -9.32187 & -9.17333 & 1740.12 \\ 4.83272 & -5.16151 & -18.6659 & 8.60083 & 1367.81 \\ -3.75725 & -4.12275 & -15.6931 & -1.27146 & 42.2184 \\ 0.822724 & -8.41263 & 8.7896 & -4.22075 & 855.35 \\ -0.66589 & -8.57498 & -10.1112 & 0.323483 & 1438.91 \\ 7.35974 & 0.413766 & 7.88215 & 10.1777 & 623.721 \\ -0.188272 & -0.997942 & 8.95715 & 2.55741 & 766.992 \\ -6.96605 & -8.93805 & -10.1383 & -9.71824 & 322.27 \\ 6.4262 & -5.24694 & 0.967467 & -6.53783 & 1925.59 \\ -1.17673 & -1.23311 & -2.4996 & -8.07426 & 1955.73 \\ 0.960791 & -4.93414 & -0.393167 & -5.67739 & 512.911 \\ 9.94929 & -7.27666 & 18.8532 & -3.3164 & -279.247 \\ -2.01168 & 6.86982 & -11.6767 & -2.38462 & -1905.41 \\ 2.9444 & -0.314675 & -8.02975 & 4.3121 & -356.85 \\ 3.41247 & -3.67489 & -14.217 & -3.99181 & -200.994 \\ 0.494627 & -2.89827 & 0.472341 & 8.49853 & 1006.07 \\ 0.0765697 & 5.69611 & 7.03799 & -2.36928 & 325.994 \\ 7.27692 & -8.61824 & -12.1684 & 4.91361 & 587.205 \\ -5.11773 & 8.19495 & -5.65902 & 9.3503 & 89.8187 \\ -6.96305 & 0.203916 & 1.1571 & 10.8467 & 1163.26 \\ 3.55407 & 4.14093 & 14.3747 & 1.63516 & 1984.67 \\ -3.82802 & -3.00717 & 8.50152 & 7.84148 & 648.395 \\ 9.86491 & 6.01187 & -16.1494 & -3.66699 & -1087.2 \end{bmatrix} \tag{3.30}$$

3.8.2 Updating the Velocity

Updating the velocity is given by:

$$v(j,t) = C[\omega \times v_(j,t) + c_1 \times rand_1 \times (pbest(j,t) - x(j,t)) + c_2 \times rand_2 \times (gbest(j,t) - x(j,t))] \tag{3.31}$$

where C is the constriction factor, *pbest* and *gbest* are the local and global best individual, ω is the inertia weight parameter which controls the global and local exploration capabilities of the particle, c_1, c_2 are cognitive and social coefficients, respectively, and $rand_1$ and $rand_2$ are random numbers betwee 0 and 1.A large cognitive coefficientc_1 means having the particle pull it to the local best position wheras a large social coefficient c_2 will pull the particle to the global position. A larger inertia weight factor is used during initial exploration its value is gradually reduced as the search proceeds. ω is given by:

$$\omega = (\omega_{max} - \omega_{min}) \times \frac{iter_{max} - iter}{iter_{max} + \omega_{min}} \tag{3.32}$$

where $iter_{max}$ is the maximum number of iterations. The constriction factor C is given by:

$$C = \frac{2}{|2 - \varphi - sqrt\varphi^2 - 4\varphi|} \tag{3.33}$$

where $4.1 \leq \varphi \leq 4.2$ [25]. The optimal value of φ was investigated by [25]. As φ increases, the factor C decreases and convergence becomes slower because population diversity is reduced. φ is given by:

$$\varphi = c_1 + c_2 \tag{3.34}$$

In updating the population, we modify an element of an individual using:

$$X_1(t, j) = X_1(t, j) + v_(t, j) \tag{3.35}$$

where $X_1(t, j)$ is an element of an individual, X_1. The method 3.35 is applied to all elements of X_1 to all individuals of the whole population.

For example, if we let $c_1 = 2$, $c_2 = 2.1$, then $\varphi = 4.1$, then we can calculate for C

$$\begin{aligned} C &= \frac{2}{|2 - \varphi - sqrt\varphi^2 - 4\varphi|} \\ C &= \frac{2}{|2 - 4.1 - \sqrt{4.1^2 - 4 \cdot 4.1}|} \\ C &= 0.7969 \end{aligned} \tag{3.36}$$

if we let $iter_{max} = 1000$ and $iter = 1$, $\omega_{min} = 0.1$, $\omega_{max} = 1.0$

$$\begin{aligned} \omega &= (\omega_{max} - \omega_{min}) \times \frac{iter_{max} - iter}{iter_{max}} + \omega_{min} \\ \omega &= (1 - 0.1) \times \frac{1000 - 1}{1000} + 0.1 \\ \omega &= 0.9991 \end{aligned} \tag{3.37}$$

Given that we have *pbest* and *gbest* with the same size as X_1, we define them by:

$$pbest = \begin{bmatrix} 10 & 9 & \dots \\ 9 & 10 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

$$gbest = \begin{bmatrix} 9 & 9 & \dots \\ 10 & 10 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

If we plug in the C, ω, and generate random numbers between 0 and 1 for $rand_1 = 0.5$ and $rand_2 = 0.5$ then we can update the velocity population

$$\begin{aligned} v(1,1) = C[\omega \times v_(1,1) + c_1 \times rand_1 \times (pbest(1,1) - x(1,1)) + c_2 \times rand_2 \times (gbest(1,1) - x(1,1))] \\ v(1,1) = 0.7969[0.9991 \times 3.83873 + 2 \times 0.5 \times (10 - 6.286) + 2.1 \times 0.5 \times (9 - 6.286)] \\ v(1,1) = 8.2869 \end{aligned} \tag{3.38}$$

we update $v(1,1)$ in the velocity vector:

$$X_1 = \begin{bmatrix} 8.2869 & -0.30638 & \dots \\ 2.83554 & 0.872582 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.39}$$

The method will be applied to all the elements of every velocity vector.

3.8.3 Updating the population

Given a velocity vector, the velocity is used to update the population. To update the individual using the velocity 3.39 and the individual 3.28:

$$\begin{aligned} X_1(t,j) &= X_1(t,j) + v(t,j) \\ X_1(1,1) &= X_1(1,1) + v(1,1) \\ X_1(1,1) &= 6.286 + 8.2869 \\ X_1(1,1) &= 14.5729 \end{aligned} \tag{3.40}$$

The newly updated individual X_1 3.28 for $X_1(1,1)$ will be

$$X_1 = \begin{bmatrix} 14.5729 & 10.888 & \dots \\ 7.025 & 6.495 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.41}$$

The method is applied to every element of every individual in the population using their respective velocity element and velocity.

After every iteration of the whole process of particle swarm optimization, the whole population undergoes constraints handling method 3.5.

3.9 Self-Organizing Hierarchical Particle Swarm Optimization with time-varying acceleration coefficients

The self-organizing hierarchical particle swarm optimization was first presented in [6] and was applied in a related scheduling problem in [28].

The main difference between the classical PSO with time-varying inertia weight approach and the SOHPSO with time-varying acceleration coefficients is the way the velocity is updated over many iterations since the velocity of a particle is the one that changes individuals.

3.9.1 Initialization of Velocity

The algorithm uses a modifying variable called velocity, which is updated and added to an individual. The individual will then undergo evaluation and will be carried on the next iteration if it has better fitness. Given an individual X_i

$$X_1 = \begin{bmatrix} & & & & \overbrace{\quad}^{P_s(i,t)} \\ 6.286 & 10.888 & 17.728 & 19.139 & 614.473 \\ 7.025 & 6.495 & 24.228 & 13.916 & 1408.858 \\ 9.248 & 8.603 & 19.135 & 19.220 & 508.199 \\ 10.853 & 13.106 & 20.974 & 15.026 & 1306.477 \\ 5.048 & 12.754 & 10.884 & 15.423 & 2140.580 \\ 10.289 & 12.332 & 23.131 & 15.012 & 1802.334 \\ 5.801 & 12.407 & 21.384 & 16.137 & 1298.278 \\ 11.635 & 6.186 & 26.206 & 16.807 & 1144.574 \\ 10.374 & 8.144 & 16.826 & 14.621 & 1117.317 \\ 5.792 & 10.457 & 27.300 & 15.581 & 1603.367 \\ 9.368 & 13.049 & 13.471 & 16.530 & 1888.533 \\ 13.561 & 13.268 & 24.568 & 20.491 & 2116.780 \\ 8.961 & 7.472 & 14.896 & 18.748 & 2169.661 \\ 10.806 & 11.287 & 10.917 & 19.594 & 732.386 \\ 7.481 & 10.335 & 23.684 & 21.219 & 2366.600 \\ 8.263 & 13.933 & 15.528 & 19.541 & 645.905 \\ 13.098 & 8.009 & 19.400 & 13.771 & 1479.827 \\ 12.416 & 13.345 & 21.114 & 16.716 & 2099.820 \\ 7.450 & 13.526 & 15.376 & 14.729 & 949.400 \\ 13.743 & 14.092 & 23.235 & 24.775 & 2124.337 \\ 14.890 & 10.207 & 28.781 & 18.824 & 540.201 \\ 14.325 & 8.338 & 21.022 & 13.481 & 1462.222 \\ 9.930 & 7.664 & 16.247 & 16.139 & 2355.188 \\ 12.175 & 13.470 & 23.892 & 22.326 & 995.791 \end{bmatrix} \tag{3.42}$$

The velocity have the same vector size as X_1. Every elements in an individual have their own velocity. The initialization of velocity is given by:

$$\begin{aligned} v_{(t,j)} &= U(-|Q_h^{max}[j] - Q_h^{min}[j]|, |Q_h^{max}[j] - Q_h^{min}[j]|) \\ v_{(t,i)} &= U(-|P_s^{max}[i] - P_s^{min}[i]|, |P_s^{max}[i] - P_s^{min}[i]|) \end{aligned} \tag{3.43}$$

where $Q_h^{max}[j]$ and $Q_h^{min}[j]$ are the maximum and minimum water discharge of the jth hydro plant. $P_s^{max}[i]$ and $P_s^{min}[i]$ are the maximum and minimum power generation

of the *ith* thermal plant. An example velocity vector is given by:

$$v_1 = \begin{bmatrix} 3.83873 & -0.30638 & -2.83883 & -0.120536 & 201.403 \\ 2.83554 & 0.872582 & -9.32187 & -9.17333 & 1740.12 \\ 4.83272 & -5.16151 & -18.6659 & 8.60083 & 1367.81 \\ -3.75725 & -4.12275 & -15.6931 & -1.27146 & 42.2184 \\ 0.822724 & -8.41263 & 8.7896 & -4.22075 & 855.35 \\ -0.66589 & -8.57498 & -10.1112 & 0.323483 & 1438.91 \\ 7.35974 & 0.413766 & 7.88215 & 10.1777 & 623.721 \\ -0.188272 & -0.997942 & 8.95715 & 2.55741 & 766.992 \\ -6.96605 & -8.93805 & -10.1383 & -9.71824 & 322.27 \\ 6.4262 & -5.24694 & 0.967467 & -6.53783 & 1925.59 \\ -1.17673 & -1.23311 & -2.4996 & -8.07426 & 1955.73 \\ 0.960791 & -4.93414 & -0.393167 & -5.67739 & 512.911 \\ 9.94929 & -7.27666 & 18.8532 & -3.3164 & -279.247 \\ -2.01168 & 6.86982 & -11.6767 & -2.38462 & -1905.41 \\ 2.9444 & -0.314675 & -8.02975 & 4.3121 & -356.85 \\ 3.41247 & -3.67489 & -14.217 & -3.99181 & -200.994 \\ 0.494627 & -2.89827 & 0.472341 & 8.49853 & 1006.07 \\ 0.0765697 & 5.69611 & 7.03799 & -2.36928 & 325.994 \\ 7.27692 & -8.61824 & -12.1684 & 4.91361 & 587.205 \\ -5.11773 & 8.19495 & -5.65902 & 9.3503 & 89.8187 \\ -6.96305 & 0.203916 & 1.1571 & 10.8467 & 1163.26 \\ 3.55407 & 4.14093 & 14.3747 & 1.63516 & 1984.67 \\ -3.82802 & -3.00717 & 8.50152 & 7.84148 & 648.395 \\ 9.86491 & 6.01187 & -16.1494 & -3.66699 & -1087.2 \end{bmatrix} \tag{3.44}$$

3.9.2 Updating the Velocity

Updating the velocity is given by:

$$\begin{aligned} v(j,t) = ((c_{1f} - c_{1i})\frac{iter}{iter_{max}} + c_{1i}) \times rand_1 \times (pbest(j,t) - x(j,t)) \\ +((c_{2f} - c_{2i})\frac{iter}{iter_{max}} + c_{2i}) \times rand_2 \times (gbest(j,t) - x(j,t)) \end{aligned} \tag{3.45}$$

if $v_id = 0$ and $rand_3 < 0.5$ then $v_{id} = rand_4 \times V_{dmax}$
else $v_{id} = -rand_5 \times V_{dmax}$

where c_{1i}, c_{1f}, c_{2i}, and c_{2f} are initial and final values of cognitive and social acceleration factors, respectively.

As an example, if $iter_{max} = 400$, $iter = 1$, $c_{1i} = 2.5$, $c_{1f} = 0.5$, $c_{2i} = 0.5$, $c_{2f} = 2.5$, $rand_1 = 0.5$, and $rand_2 = 0.5$ is used in updating $v(1,1)$ Given that we have *pbest* and *gbest* with the same size as X_1, we define them by:

$$pbest = \begin{bmatrix} 10 & 9 & \dots \\ 9 & 10 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

$$gbest = \begin{bmatrix} 9 & 9 & \dots \\ 10 & 10 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

To update velocity:

$$v(j,t) = ((c_{1f} - c_{1i})\frac{iter}{iter_{max}} + c_{1i}) \times rand_1 \times (pbest(j,t) - x(j,t)) + ((c_{2f} - c_{2i})\frac{iter}{iter_{max}} + c_{2i}) \times rand_2 \times (gbest(j,t) - x(j,t))$$
$$v(1,1) = ((0.5 - 2.5)\frac{1}{400} + 2.5) \times 0.5 \times (10 - 6.286) + ((2.5 - 0.5)\frac{1}{400} + 0.5) \times 0.5 \times (9 - 6.286)$$
$$v(1,1) = 5.3185 \tag{3.46}$$

The new $v(1,1)$ will be updated:

$$v_1 = \begin{bmatrix} 5.3185 & 10.888 & \dots \\ 7.025 & 6.495 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix}$$

3.9.3 Updating the population

Given a velocity vector, the velocity is used to update the population. To update the individual using the velocity 3.39 and the individual 3.28:

$$\begin{aligned} X_1(t,j) &= X_1(t,j) + v(t,j) \\ X_1(1,1) &= X_1(1,1) + v(1,1) \\ X_1(1,1) &= 6.286 + 8.2869 \\ X_1(1,1) &= 14.5729 \end{aligned} \tag{3.47}$$

The newly updated individual X_1 3.28 for $X_1(1,1)$ will be

$$X_1 = \begin{bmatrix} 14.5729 & 10.888 & \dots \\ 7.025 & 6.495 & \dots \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{3.48}$$

The method is applied to every element of every individual in the population using their respective velocity element and velocity.

After every iteration of the whole process of particle swarm optimization, the whole population undergoes constraints handling method 3.5.

3.10 Plan of Operations

Plan of operations section shows a more detailed explanation and process of the proposed hybrid algorithm. There are initial paramaters that should be considered before the program will run.

Method	N_p	P_e	P_c	P_m	*step*	*visual*	$\varepsilon_{vcourse}$	ε_{vfine}	$\varepsilon_{pcourse}$	ε_{pfine}	$iteration_{course}$	$iteration_{fine}$
RCGA	100	30	30	40	-	-	1	0.01	2	0.01	20	10
RCGA-AFSA	100	30	30	40	0.05	1.0	1	0.01	2	0.01	20	10
RCGA-PSO	100	30	30	40	-	1.0	1	0.01	2	0.01	20	10
RCGA-SOHPSO	100	30	30	40	-	1.0	1	0.01	2	0.01	20	10

Table 3.3: Example parameters

There are $N_p = 100$ number of individuals in a population, where an individual in N_p is X_{n_p}. An example individual will be X_1 which when expanded is:

$$X_1 = \begin{bmatrix}
11.4457 & 6.3 & 29.8281 & 17 & 108.948 & 161.872 & 138.093 \\
7.90411 & 7.89401 & 17.9337 & 16.3347 & 79.7753 & 155.76 & 142.592 \\
7.83467 & 8.7975 & 18.8026 & 16.1203 & 66.7125 & 93.5818 & 117.536 \\
7.95456 & 8.137 & 18.6841 & 15.6137 & 73.7743 & 69.0391 & 89.4371 \\
7.70685 & 8.52737 & 18.4447 & 16.9329 & 67.5078 & 85.0032 & 88.0706 \\
7.80512 & 8.23525 & 17.8995 & 16.1731 & 70.6062 & 142.867 & 159.347 \\
7.993 & 8.43823 & 18.0994 & 15.9354 & 109.637 & 208.026 & 204.085 \\
8.02112 & 8.08024 & 18.7906 & 16.6475 & 128.269 & 194.582 & 256.503 \\
7.90535 & 8.66066 & 17.9437 & 16.5988 & 134.826 & 256.333 & 261.938 \\
8.40289 & 8.7109 & 18.3996 & 16.5758 & 131.644 & 245.303 & 260.723 \\
8.01889 & 8.56133 & 17.4743 & 16.7956 & 135.226 & 227.08 & 293.253 \\
8.13231 & 8.75444 & 17.7688 & 16.3982 & 143.025 & 241.807 & 319.055 \\
8.47342 & 8.09286 & 17.1495 & 16.9532 & 137.438 & 210.433 & 310.432 \\
8.25096 & 8.09474 & 17.9236 & 16.9711 & 139.959 & 224.644 & 214.291 \\
7.8689 & 8.69737 & 18.0079 & 16.8108 & 94.2826 & 188.286 & 274.976 \\
8.03669 & 8.71759 & 17.1329 & 17.3604 & 138.825 & 241.225 & 219.504 \\
8.5453 & 8.53868 & 17.4691 & 17.0107 & 138.393 & 205.638 & 245.449 \\
7.79056 & 8.47567 & 17.3968 & 17.567 & 112.732 & 226.431 & 324.004 \\
8.03784 & 8.36339 & 17.5608 & 17.4678 & 137.478 & 237.039 & 238.697 \\
8.65221 & 8.75875 & 17.3387 & 17.6087 & 132.227 & 223.693 & 230.948 \\
7.96704 & 8.30456 & 17.7798 & 18.0351 & 86.3067 & 149.041 & 216.753 \\
7.82883 & 8.56465 & 17.1123 & 17.7663 & 102.307 & 145.971 & 153.839 \\
7.83084 & 8.6299 & 16.856 & 17.6053 & 71.2624 & 135.504 & 185.259 \\
7.90613 & 8.51025 & 17.2547 & 17.8531 & 62.3249 & 148.186 & 132.609
\end{bmatrix}$$

where $N_h = 4$ and $N_s = 3$.

Chapter 4

Results and Discussion

The parameters in Section 3.1 is used in their respective algorithms.

4.1 Methods

4.1.1 RCGA

The pure RCGA method was ran 30 times to attain a credible solution. In test system 1 for all cases, a total of 1000 iterations is used [19]. In test system 2 for all cases, a total of 1400 iterations is used.

4.1.2 RCGA-AFSA

The hybrid RCGA-AFSA method was ran 30 times. Iterations for test system 1 in all cases of the RCGA and AFSA is 600 and 400, respectively. In test system 2 for all cases, a total of 1000 for RCGA and 400 for AFSA iterations are used.

4.1.3 RCGA-PSO

The hybrid RCGA-PSO method was ran 30 times. Iterations for test system 1 in all cases of the RCGA and PSO is 600 and 400, respectively. In test system 2 for all cases, a total of 1000 for RCGA and 400 for PSO iterations are used.

4.1.4 RCGA-SOHPSO

The proposed hybrid RCGA-PSO method was ran 30 times. Iterations for test system 1 in all cases of the RCGA and SOHPSO is 600 and 400, respectively. In test system 2 for all cases, a total of 1000 for RCGA and 400 for SOHPSO iterations are used.

4.2 Results

system	case	method	ave. best fitness	ave. worst fitness
1	1	RCGA	908215	940105
1	1	RCGA-AFSA	904338	927770
1	1	RCGA-PSO	914646	944457
1	1	RCGA-SOHPSO	892154	936900
1	2	RCGA	917760	951885
1	2	RCGA-AFSA	914890	940596
1	2	RCGA-PSO	925355	955162
1	2	RCGA-SOHPSO	899068	946753
2	1	RCGA	44926.8	49811.5
2	1	RCGA-AFSA	45842.2	49590.3
2	1	RCGA-PSO	46442.9	76115.4
2	1	RCGA-SOHPSO	46320.8	54066.6
2	2	RCGA	45818.6	49850.9
2	2	RCGA-AFSA	44490.7	49972.7
2	2	RCGA-PSO	46455.1	51480.4
2	2	RCGA-SOHPSO	46333.8	51146.1

The results that can be seen in Figure 4.2 show the methods for test systems with their average best fitness and average worst fitness. All things being equal in their respective test systems and cases, the proposed algorithm RCGA-SOHPSO shows an improvement on some test systems and particular cases. The hybrid algorithm RCGA-SOHPSO yields the best result in test system 1 in all cases in contrast with RCGA, RCGA-AFSA, and RCGA-PSO. The hybrid algorithm RCGA-PSO performed the worst in all test systems and their respective cases when it comes to the average best fitness. In test system 2 case 2, the RCGA-AFSA method performs the best when it comes to the average best fitness.

The RCGA-SOHPSO yields satisfactory average best fitness when it comes to test system 2 in all cases. It can be deduced that RCGA-SOHPSO works well in less complex

multiobjective, dynamic, nonlinear optimizations. The RCGA-AFSA yields better result when it comes to optimization problems that are complex since its implementation can handle multiple variables at the same time. The AFSA method in the RCGA-AFSA performs well in complex systems since an individual is treated and updated as one unlike the PSO method in the RCGA-PSO and SOHPSO in RCGA-SOHPSO wherein every element in every individual have their own method of updating themselves using their velocity.

The idea of having an individual move as one entity in the solution search space is better since every element of an individual is dependent with each other by equality constraints unlike when the individual's element move in their own. Having the elements move on their own disregards their property wherein all the elements are connected and dependent to each other. An example of this relationship with each element is the equality constraint final water volume reservoir with the water discharge rate of a particular hydro plant in all time interval. If there is an overall surplus of water in the reservoir, the whole elements of the particular hydro plant moves as one consuming the water surplus in the reservoir. In the case wherein elements move on their own, One element can deviate from the general movement of all the other elements. An example of this case is when there is a surplus of water in the reservoir and the general direction of the population is consuming the surplus. But if one element deviate from the direction of consuming the surplus, where the idea of the SHS problem is an optimal maximization of the water, then the power generation of the thermal plant will be affected and thus will yield a less efficient set of solutions because the thermal plant will consume more fuel because of the higher power demand for the thermal plant.

Swarming algorithms loses their potency in finding a good solution when the system becomes more complex and when individuals posses more variables or set of solutions[41]. The RCGA will still function and does not lose its good searching capability since finding good solutions does not involve other variables to greatly affect an individual. The only downside of using a pure RCGA is the complexity of implementing it and the necessary memory and CPU usage. The hybrid algorithm of RCGA-AFSA, RCGA-PSO, and RCGA-SOHPSO lessen the supposed longer complexity runtime of the RCGA assuming ceteris paribus.

4.2.1 Reservoir Volume Violations

system	case	method	Reservoir 1	Reservoir 2	Reservoir 3	Reservoir 4	Total
1	1	RCGA	0.05%	0.27%	0.35%	0.13%	0.8%
1	1	RCGA-AFSA	0.20%	0.32%	0.06%	0.18%	0.76%
1	1	RCGA-PSO	0.30%	1.16%	0.004%	0.35%	1.814%
1	1	RCGA-SOHPSO	0.69%	0.28%	0.35%	0.29%	1.61%
1	2	RCGA	0.0052%	0.43%	0.15%	0.54%	1.64%
1	2	RCGA-AFSA	0.21%	0.32%	0.04%	0.26%	0.83%
1	2	RCGA-PSO	0.81%	0.80%	0.09%	0.14%	1.85%
1	2	RCGA-SOHPSO	0.09%	0.22%	0.06%	0.04%	0.41%
2	1	RCGA	0.15%	1.33%	0.82%	0.04%	2.70%
2	1	RCGA-AFSA	0.38%	0.41%	0.20%	0.20%	1.19%
2	1	RCGA-PSO	0.42%	0.88%	0.32%	1.00%	2.62%
2	1	RCGA-SOHPSO	0.08%	0.23%	0.06%	0.28%	0.65%
2	2	RCGA	0.05%	1.18%	0.51%	0.05%	1.79%
2	2	RCGA-PSO	0.29%	0.87%	0.33%	0.93%	2.42%
2	2	RCGA-SOHPSO	0.38%	1.01%	0.04%	0.55%	1.98%

All the methods generated optimal reservoir storage volume and satisfied all the hydraulic and electric constraints. The percent error or violation fell into small values with the RCGA-SOHPSO method in Test system 1 Case 2 having the smallest percent error.

Chapter 5

Conclusion and Recommendation

The use of RCGA and hybrids of RCGA-AFSA, RCGA-PSO, and RCGA-SOHPSO with their appropriate parameters given on empiricial findings [42, 36, 19, 38, 41, 25, 6, 28] and in 3.1 can successfully find solutions that are satisfactory and at times excellent since the stochastic nature of these algorithms makes them good in finding solutions for complex problems. Additional research for these hybrids is required in this particular problem since hybrids of algorithm are unusual in solving the SHS problem.

It is concluded that all the algorithms give satisfactory and sometimes excellent solutions. It is also suggested that dealing with less complex systems, (e.g. test system 1), the hybrid of RCGA-SOHPSO is a strong candidate to use in minimization of the SHS problem. However, in dealing with more complex systems, (e.g. test system 2), the RCGA and RCGA-AFSA yields more excellent results.

5.0.2 Recommendation

Future study and research on how different hybrid algorithms affect complex problems will further the development and understanding the nature of stochastic methods. Finding new parameters and methods, e.g. the SBX in the crossover operator and Michalewicz's Non-Uniform Mutation in the mutation operator of the real coded genetic algorithm, the visual distance and step size of the artificial fish swarm algorithm, and the method of updating the velocity of the particle swarm optimization, that will suit particular systems can also help the objective of the SHS problem be more minimized and excellent.

List of References

[1] http://www.leander-project.homecall.co.uk/Engines/turbine.jpg.

[2] http://philschatz.com/physics-book/resources/Figure_24_05_01.jpg.

[3] http://img.bhs4.com/FA/8/FA89BF0370B42A4719540EFC583EFDDD331BFD22_lis.jpg.

[4] http://upload.wikimedia.org/wikipedia/commons/9/93/Water_turbine_runners.jpg.

[5] D. S. 0. Nilsson, *Mixed-integer programming applied to short-term planning of a hydro-thermal system*, IEEE Transactions on Power Systems, 11 (1996), pp. 281–286.

[6] H. W. A. Ratnaweera, S.K. Halgamuge, *Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients*, IEEE Transactions on Evolutionary Computation, 8 (2004), pp. 240–255.

[7] D. F. A.P. Alves Da Silva, *Fundamentals of Genetic Algorithms*, Institute of Electrical aand Electronics Engineers, Inc, 2008.

[8] Y. Y. A.P. Georgakakos, H. Yao, *A control model for dependable hydropower capacity optimization*, Water Resources Research, 33 (1997), pp. 2349–2365.

[9] M. Basu, *Improved differential evolution for short-term hydrothermal scheduling*, Electrical Power and Energy Systems, 58 (2013), pp. 91–100.

[10] ——, *Improved differential evolution for short-term hydrothermal scheduling*, Electrical Power and Energy Systems, 58 (2014), pp. 91–100.

[11] C. C.-H. F. I.-K. L. P. B. Chang, Shi-Chung, *Hydroelectric generation scheduling with an effective differential dynamic programming algorithm*, IEEE Transactions on Power Systems, 5 (1990), pp. 737–743.

[12] D. L. S. Chao-an Li, Philip J. Jap, *Implementation of network flow programming to the hydrothermal coordination in an energy management system*, IEEE Transactions on Power Systems, 8 (1993), pp. 1045–1053.

[13] H. B. D Sjelvgren, *Large-scale non-linear programming applied to operations planning*, IEEE Transactions on Power Systems, 11 (1989), pp. 213–217.

[14] A. L. R. S. D.A.G. Vieira, L.S.M. Guedes, *Formulations for hydroelectric energy production with optimality conditions*, Energy Conversion and Management, 89 (2015), pp. 781–788.

[15] G. S. D.C. Walters, *Genetic algorithm solution of economic dispatch with valve point loading*, IEEE Transactions on Pwer Systems, 8 (1993), pp. 1325–1332.

[16] K. Deb, *An efficient constraint handling method for genetic algorithms*, Computer Methods in Applied Mechanics and Engineering, 168 (2000), pp. 311–338.

[17] K. Deb and R. B. Agrawal, *Simulated binary crossover for continuous search space*, Complex Systems, 9 (1995), pp. 115–148.

[18] K. Deep and M. Thakur, *A new mutation operator for real coded genetic algorithms*, Applied Mathematics and Computation, 193 (2007), pp. 211–230.

[19] N. Fang, J. Zhou, R. Zhang, Y. Liu, and Z. Y, *A hybrid of real coded genetic algorithm and artificial fish swarm algorithm for short-term optimal hydrothermal scheduling*, Electrical Power and Energy Systems, 62 (2014), pp. 617–629.

[20] R. N. K. S. G. Kumar, V. Shaarma, *Quadratic migration of biogeography based optimization for short term hydrothermal scheduling*, in First International Conference on Networks & Soft Computing, 2014.

[21] J. G. W. J. M. T. B. S. R. M. C. Gary W. Chang, Mohamed Aganagic, *Experiences with mixed integer linear programming based approaches on short-term hydro scheduling*, IEEE Transactions on Power Systems, 16 (2001), pp. 743–749.

[22] S. S. G.G. Oliveira, *A second-order network flow algorithm for hydrothermal scheduling*, IEEE Transactions on Power Systems, 10 (1995), pp. 1653–1641.

[23] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Publishing Company, Readling, Massachusetts, 1989.

[24] R. E. J. Kennedy, *Particle swarm optimization*, 1995.

[25] K. L. J.G. Vlachogiannis, *A comparative study on particle swarm optimization for optimal steady-state performance of pwer systems*, IEEE Transactions on Power SYstems, 21 (2006), pp. 1718–1728.

[26] J. Klein, tech. report.

[27] F. Knopf, *Modeling Analysis and Optimization of PRocess and Energy Systems*, John Wiley & Sons, Inc., Hoboken, New Jersey, 2012.

[28] L. S. K.T. Chaturvedi, M. Pandit, *Self-organizing hierarchical particle swarm optimization for nonconvex economic dispatch*, IEEE Transactions on Power Systems, 23 (2008), pp. 1079–1087.

[29] M. R. P. S. L. Bayon, J.M. Grau, *Influence of the elevation-storage curve in the optimization of hydro-plants*, in Second Internation Conference on Multidisciplinary Design Optimization and Applications, 2008.

[30] S. S. L. Lakshminarasimman, *A modified hybrid differential evolution for short-term scheduling of hydrothermal power systems with cascaded reservoirs*, Energy Conversion and Management, 49 (2008), pp. 2513–2521.

[31] X. Li, Z. Shao, and J. Qian, *An optimizing method based on autonomous animats: fish-swarm algorithm*, System Engineering: THeory and Practice, 22 (2002), pp. 32–38.

[32] A. R. H. Md. Sayeed Salam, Khalid Mohamed Nor, *Hydrothermal scheduling based lagrangian relaxation approach to hydrothermal coordination*, IEEE Transactions on Power Systems, 13 (1998), pp. 226–235.

[33] G. C. M.E. El, Hawary, *Optimal Economic Operation of Electric Power Systems*, vol. 142, Academic Press, Inc., 1979.

[34] M. K. M.E. El-Hawary, *Optimal parameter estimation for hydro-plant performance models in economic operation studies*, IEEE Transactions on Power Systems, PWRS-1 (1986), pp. 126–131.

[35] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, New York, 1992.

[36] A. Moradi, Y. Alinejad-Beromi, and K. Kiani, *Artificial fish swarm algorithm for solving the economic dispatch with valve-point effect*, International Journal of Engineering and Technology sciences, 2 (2014), pp. 299–313.

[37] T. L. M.R. Piekutowski, *Optimal short-term scheduling for a large-scale cascaded hydro system*, Energy Conversion and Management, 9 (1994), pp. 805–811.

[38] H. S. N. Amjady, *Daily hydrothermal generation scheduling by a new modified adaptive particle swarm optimization technique*, Electric Power Systems Research, 80 (2010), pp. 723–732.

[39] J. R. N.V. Arvanitidis, *Composite representation of a multi-reservoir hydroelectric power system*, IEEE Trans. Power Appar. Syst., (1970), pp. 319–325.

[40] J. Petilla, *Philippine energy plan 2012-2030*, 2012.

[41] Y. S. R.C. Eberhart, *Tracking and optimizing dynamic systems with particle swarms*, Proc. IEEE Congr. Evolutionary Computation 2001, (2001), pp. 94–97.

[42] R. N. S. Kumar, *Efficient real coded genetic algorithm to solve the non-convex hydrothermal scheduling problem*, Electrical Power & Energy Systems, 29 (2007), pp. 738–747.

[43] M. I. S.O. Orero, *A genetic algorithm modelling framework and solution technique for short term optimal hydrothermla scheduling*, IEEE Transactions on Power Systems, 13 (1998), pp. 501–518.

[44] R. N. Sushil Kumar, *Efficient real coded genetic algorithm to solve the non-convex hydrothermal scheduling problem*, Electrical Power and Energy Systems, 29 (2007), pp. 738–747.

[45] L. M. R. Z. Y. Z. Y. Wang, J Zhou, *Short-term hydrothermal generation scheduling using differential real-coded quantum-inspired evolutionary algorithm*, Energy, 44 (2012), pp. 657–671.

[46] L. M. S. O. Y. Z. Y. Wang, J Zhou, *A clonal real-coded quantum-inspired evolutionary algorithm with cauchy mutation for short-term hydrothermal generation scheduling*, Electrical Power and Energy Systems, 43 (2012), pp. 1228–1240.

[47] C. N. Yang, Jin-Shyr, *Short term hydrothermal coordination using multi-pass dynamic programming*, IEEE Transactions on Power Systems, 4 (1989), pp. 1050–1056.

Appendix A

Table for Test System 1, Case 1

Table A.1: Test System 1, Case1, RCGA: Average Best Chromosome,Average best fitness = 908215, Average worst fitness = 940105, CPU time = 608.35 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$
1	15	12.9263	29.9577	21.8	932.582
2	9.48048	8.79973	24.6666	17.1918	1002.82
3	9.40223	8.6869	24.747	17.7997	956.215
4	9.1585	9.58317	24.4589	17.4741	875.664
5	8.74023	9.08706	23.7299	18.2194	866.937
6	9.40315	9.28054	24.4043	17.9574	984.542
7	8.59027	9.59385	24.1263	18.261	1220.93
8	8.8855	8.92495	23.6752	18.5784	1564.64
9	8.78331	8.8504	21.0627	18.8274	1787.06
10	8.35382	8.65412	21.2143	17.7423	1873.48
11	8.47631	8.46896	19.9939	18.1119	1772.14
12	7.89156	8.57256	17.4413	18.4813	1839.32
13	8.08762	8.59159	19.1883	18.8516	1758.73
14	7.59132	8.5947	16.9398	19.1553	1723.18
15	7.62533	8.24866	16.1754	18.8885	1651.12
16	7.42448	7.61141	16.0639	18.1278	1602.22
17	7.57741	8.23559	15.1223	17.2865	1660.65
18	7.27118	8.25611	13.7879	17.463	1670.18
19	6.83093	8.09518	13.3884	17.4183	1775.82
20	7.27769	7.89303	13.2801	17.4311	1816.35
21	6.98389	7.53806	13.0241	17.6429	1780.15
22	7.22675	7.31413	12.7993	17.4804	1664.6
23	6.80076	7.59957	12.549	17.4664	1398.19
24	6.72602	7.35159	12.5091	17.2727	1144.54

Table A.2: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	98.5	79.5404	0 259.378	
2	82.0005	65.1166	15.1499	224.914
3	81.2426	64.7611	12.0995	245.682
4	78.6054	68.879	14.4774	252.374
5	75.6786	65.8979	17.4716	264.015
6	77.9542	65.0941	12.5244	269.885
7	74.411	64.0841	13.2935	277.28
8	75.4821	59.723	14.2836	285.873
9	75.8836	58.5037	25.5293	293.025
10	74.1226	57.599	23.7891	291.011
11	75.9887	57.3791	28.2616	296.226
12	73.3449	57.3739	38.7641	301.199
13	75.7303	56.8356	33.2216	305.477
14	73.7788	57.0704	40.6829	305.284
15	74.7058	56.191	44.4786	303.503
16	73.3361	52.9021	43.7581	297.786
17	75.1679	55.6398	47.6435	290.904
18	73.0887	53.5996	53.0355	290.095
19	70.0437	52.3086	53.8781	287.954
20	72.7354	51.3893	54.7579	284.768
21	70.9809	50.5689	56.17	282.126
22	72.7317	50.477	55.5652	276.631
23	69.9827	52.3251	57.879	271.622
24	69.	8272 51.3912	58.282	265.956

Table A.3: Test System 1, Case1, RCGA-AFSA: Average Best Chromosome,Average best fitness = 904338, Average worst fitness = 927770, CPU time = 967.83 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$
1	11.6451	6.38398	29.8169	16.8624	995.211
2	9.19093	8.42242	25.1455	15.1662	1020.93
3	8.65311	8.95938	24.9224	15.5147	970.509
4	8.52875	8.8926	24.9735	16.0586	888.267
5	8.7658	8.76511	24.0274	16.226	875.384
6	9.02322	8.69029	24.076	16.4679	989.772
7	8.53888	8.39225	22.5133	16.202	1221.54
8	9.19628	8.61948	22.9764	16.4607	1562.04
9	8.52975	8.39375	21.155	17.1374	1790.27
10	8.20739	8.45748	19.987	17.7722	1856.19
11	8.43665	8.51886	19.6996	17.8754	1758.15
12	8.18021	8.87167	18.8085	18.0198	1832.32
13	8.44196	8.4631	17.5034	18.3612	1743.96
14	8.52877	8.63879	16.6883	18.5585	1705.14
15	8.39913	8.44682	16.3587	18.6765	1634.9
16	8.28023	8.45362	14.4485	17.9906	1574.5
17	8.80493	8.43332	13.8259	19.1608	1624.57
18	7.98454	8.5089	13.3131	18.6645	1646.73
19	7.60723	8.00339	13.3291	19.1267	1752.51
20	8.15138	7.90346	12.8867	18.8364	1795.45
21	7.68274	8.19453	12.5991	19.5376	1756.92
22	7.02477	8.36086	12.5479	19.1082	1648.43
23	7.29938	8.94494	13.27	19.4261	1379.48
24	6.42959	6.95539	12.983	15.372	1165.12

Table A.4: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	85.2723	50.8528	0.0733847	238.591
2	80.9358	63.5222	10.5158	214.433
3	78.037	66.344	8.94312	232.796
4	76.7963	66.2874	8.86602	245.821
5	76.7437	65.1391	12.5536	256.029
6	77.4013	63.6986	9.1285	265.779
7	74.6817	60.3958	16.7839	270.859
8	77.9249	60.2424	14.4678	279.63
9	75.0382	58.6224	20.8422	289.819
10	74.0784	59.5602	26.2317	298.612
11	76.439	60.2876	26.5881	303.286
12	75.7029	61.7258	29.7545	305.31
13	77.1951	59.2865	35.3351	309.484
14	78.6371	60.2294	40.1533	312.498
15	78.4775	59.4957	41.4515	311.971
16	78.6873	59.7514	48.2109	305.59
17	81.5726	58.4731	50.4104	311.679
18	76.3722	56.7392	52.4322	303.492
19	74.1972	53.4535	52.9537	301.971
20	76.6521	52.8044	54.8263	296.438
21	73.8996	55.2988	56.1973	294.54
22	69.8876	56.5197	56.8953	286.274
23	71.6691	58.5236	57.2149	281.973
24	66.079	48.7197	58.2494	251.226

Table A.5: Test System 1, Case1, RCGA-PSO: Average Best Chromosome,Average best fitness = 914646, Average worst fitness = 944457, CPU time =984.95 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$
1	7.66667	6 30	18.1961	1016.52	
2	8.93288	8.43355	24.4008	16.3073	1019.2
3	8.82838	9.41099	25.1283	15.4531	978.702
4	8.67524	8.27443	22.7708	15.1004	896.715
5	6.89222	9.17242	22.1625	16.129	885.094
6	9.12818	8.34717	24.3808	15.6903	1003.11
7	9.00078	8.0849	21.2864	16.1005	1231.48
8	8.42444	8.70812	20.6975	16.4229	1570.39
9	6.63647	7.78007	17.949	15.6546	1813.83
10	8.63358	8.68937	18.7385	16.8719	1871.13
11	7.99468	8.17498	17.6002	16.5282	1781.86
12	7.63173	8.82607	18.3079	18.6971	1844.48
13	8.49307	8.92714	15.2374	17.6163	1759.49
14	7.84695	8.30478	14.2308	16.5089	1737.97
15	8.06341	7.81942	16.0834	17.3316	1667.65
16	7.74375	8.7579	14.2171	17.7555	1597.24
17	8.05614	8.57941	16.4202	18.4094	1662.91
18	8.16153	8.52794	13.5056	17.1433	1674.04
19	7.9505	8.9054	14.2814	17.3794	1775.61
20	7.89029	7.85248	14.3103	19.0011	1815.09
21	7.36869	8.16074	13.5834	18.6785	1780.62
22	8.20242	8.5388	13.7433	17.7838	1661.24
23	7.39804	7.83813	13.9343	18.7359	1399.8
24	7.44841	8.57034	14.7864	17.7915	1144.77

Table A.6: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	64.7667	49	0	239.715
2	75.4188	61.5688	15.2163	218.762
3	74.8176	66.6859	10.1171	222.858
4	74.3617	61.0808	19.2327	230.742
5	63.8829	66.2283	21.1595	244.136
6	74.1284	60.0861	14.4649	244.39
7	72.8377	57.9368	21.4874	252.58
8	69.9	59.2163	24.6077	262.964
9	62.9611	53.6926	32.7541	259.395
10	73.6852	58.5808	27.8873	272.73
11	71.9404	55.7654	30.8202	271.603
12	70.2967	59.0078	29.7478	291.167
13	75.8036	59.3382	40.4357	281.333
14	71.7224	56.898	45.1385	273.592
15	73.982	54.9855	40.2131	277.436
16	73.867	58.5532	46.6916	279.878
17	74.5519	56.8728	41.3072	280.623
18	76.0756	54.9473	51.7721	270.035
19	74.4124	56.6355	48.2022	273.109
20	74.2309	50.7914	47.1764	279.502
21	69.9168	52.8793	49.8154	277.289
22	75.9333	55.3863	51.5919	269.598
23	71.9532	52.5928	48.5603	276.127
24	72.9538	55.9629	50.0731	266.098

Table A.7: Test System 1, Case1, RCGA-SOHPSO: Average Best Chromosome,Average best fitness = 892154, Average worst fitness = 936900, CPU time =964.5 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$
1	15	6.05469	27.695	22.1413	880.567
2	7.59291	8.53212	19.0496	16.5748	895.055
3	7.83584	8.40312	20.0429	16.52	821.994
4	8.11276	8.32645	19.5472	16.4261	758.624
5	7.97659	8.48101	20.3098	16.7696	821.426
6	7.89978	8.60752	18.3562	16.611	908.689
7	7.7797	8.35565	17.3595	16.4541	1164.78
8	7.783	8.49704	17.2661	16.5313	1516.01
9	7.65103	8.60833	19.3983	16.7799	1793.85
10	7.68524	8.59214	16.8654	16.7145	1862.09
11	7.76963	8.42805	18.4071	16.5435	1790.18
12	7.75322	8.49132	17.2852	16.8184	1847.64
13	7.90773	8.45328	16.8008	16.7289	1746.34
14	7.93128	8.27517	16.8037	16.7279	1747.87
15	7.73409	8.32376	17.2339	16.8386	1636.87
16	7.56547	8.27057	16.8099	16.6487	1602.72
17	7.68542	8.57331	17.1528	16.5815	1656.77
18	7.73122	8.2606	16.9157	16.9348	1677
19	7.84216	8.34932	17.0691	16.8513	1778.52
20	7.52101	8.2195	16.7511	16.9632	1823.47
21	7.86994	8.46484	16.6513	16.6388	1766.7
22	7.84642	8.60243	16.8466	17.0097	1642.51
23	7.7732	8.56203	17.2595	16.8203	1372.07
24	7.83821	8.49332	17.0043	16.8502	1058.6

Table A.8: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	98.5	49.3789	8.04779	262.425
2	72.4335	64.6593	41.9192	221.868
3	73.977	64.3891	39.0128	236.915
4	74.9109	64.2877	40.6358	237.354
5	73.8258	64.6335	35.6306	241.3
6	72.9399	64.5182	45.4569	241.991
7	72.4482	61.6092	50.0978	243.416
8	72.3581	61.4444	50.6178	245.149
9	72.4255	61.7616	41.4386	247.604
10	72.9967	62.0259	52.1314	248.466
11	74.8697	61.6157	47.2337	249.151
12	75.3415	61.659	50.5791	251.321
13	76.7783	61.246	52.7369	252.333
14	77.7292	60.6229	53.3359	253.178
15	77.0023	61.2545	52.1248	255.506
16	76.0219	60.7247	53.9894	255.785
17	76.9526	61.2323	52.7247	256.856
18	77.1923	57.94	54.3912	259.493
19	77.7927	57.4402	53.8236	260.502
20	75.467	56.4033	54.5147	261.885
21	77.6369	58.0504	54.9764	260.037
22	77.326	59.0432	54.7103	263.757
23	77.3359	58.5299	53.4084	263.206
24	77.9879	57.9456	53.7882	264.314

Appendix B

Table for Test System 1, Case 2

Table B.1: Test System 1, Case2, RCGA: Average Best Chromosome,Average best fitness = 917760, Average worst fitness = 951885, CPU time = 607.21 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$
1	15	13.5	29.9956	22.1901	927.956
2	9.53715	9.46286	25.4421	17.4703	1003.79
3	8.85241	8.99857	25.5402	17.3255	965.29
4	9.13329	9.50518	24.6621	17.374	877.241
5	8.69557	9.22553	23.8896	17.8936	868.549
6	8.38075	9.09913	24.5698	18.4365	985.237
7	8.34271	9.02392	23.5774	18.2653	1221.2
8	9.09513	9.14095	21.9628	18.4519	1555.15
9	8.79134	8.83086	22.5544	18.1944	1796.88
10	8.2174	8.46141	21.1883	18.1259	1874.35
11	8.07119	8.84101	19.2542	18.2894	1768.26
12	7.99061	7.99417	18.9193	18.1254	1848.57
13	8.00592	8.59811	18.7076	18.8105	1758.6
14	8.22967	8.66773	16.0855	19.4092	1713.37
15	8.03247	7.71345	14.8939	17.9496	1651.4
16	7.61696	7.91822	14.8757	17.9625	1592.4
17	7.28126	8.24183	14.8328	17.699	1658.17
18	7.58351	7.74932	13.9866	18.1674	1666.44
19	7.38322	8.1814	13.7753	18.2036	1768.17
20	6.97144	7.8719	13.4468	17.5633	1819.6
21	7.32282	7.65858	13.0998	17.5613	1780.37
22	7.33287	7.85102	13.5499	17.1823	1666.3
23	7.0031	7.41534	12.946	17.4439	1399.57
24	6.87809	7.37546	12.6615	16.8011	1146.93

Table B.2: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	98.5	82	0	261.544
2	82.1299	67.7582	10.3103	226.007
3	77.6573	65.5662	8.58092	242.906
4	78.5725	67.324	14.0563	252.806
5	75.5213	65.4417	16.4794	264.009
6	73.9683	63.5644	13.0792	274.151
7	73.3499	61.0886	14.991	279.369
8	77.0879	59.8613	21.0995	286.804
9	76.1736	58.0011	18.5739	290.371
10	73.3649	56.3108	22.7973	293.18
11	74.506	58.6634	30.1164	298.456
12	74.6699	54.2548	32.4016	300.107
13	75.5209	56.5378	33.753	305.585
14	77.7985	57.2658	42.9068	308.662
15	77.1771	53.5832	47.46	300.381
16	75.1681	54.8143	49.6047	298.013
17	73.1338	55.2016	49.6104	293.889
18	75.1257	51.5935	52.9303	293.907
19	73.897	52.9524	54.1155	290.865
20	70.72	51.2492	55.1135	283.315
21	73.2565	51.3782	56.3235	278.673
22	73.1069	53.0066	54.9358	272.654
23	71.5996	51.1925	58.0327	269.607
24	70.9192	51.3783	58.7145	262.062

Table B.3: Test System 1, Case2, RCGA-AFSA: Average Best Chromosome,Average best fitness = 914890, Average worst fitness = 940596, CPU time =959.84 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$
1	12.0219	6.06809	29.3471	16.7049	958.237
2	8.92026	9.51078	24.6233	16.078	1007.01
3	9.05348	9.28401	24.7441	16.3517	963.409
4	8.5176	9.18371	24.0532	16.5451	884.125
5	9.06856	8.82575	24.0395	16.5154	876.49
6	8.42449	8.83265	23.4391	17.3133	986.441
7	8.67482	8.68323	22.8779	17.2195	1220.05
8	8.82867	8.48493	22.3198	17.1806	1563.78
9	8.59346	8.85276	21.3849	18.1318	1789.14
10	8.62358	8.60682	20.1445	17.727	1862.9
11	8.47187	8.49266	20.1626	18.1507	1767.42
12	8.49409	8.57041	18.7658	18.1988	1839.61
13	8.29024	8.60301	17.135	18.2847	1749.13
14	7.91589	8.38378	16.6997	18.6132	1716 37
15	7.85059	8.14506	14.9584	18.1411	1644.59
16	7.57943	8.172	14.7881	18.5688	1583.34
17	7.68824	8.16525	14.6947	17.4738	1652.6
18	8.02809	8.81234	13.6123	18.179	1654.45
19	7.43994	8.30989	12.9762	17.9824	1765.37
20	7.15932	7.80666	12.7989	17.7692	1814.2
21	7.27468	8.02852	12.9162	18.2123	1772.25
22	7.27057	8.34477	12.7176	17.6662	1658.7
23	7.04803	7.73779	12.755	18.5038	1391.49
24	6.0834	6.53128	13.1753	14.6714	1174.58

Table B.4: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	89.6176	49.4417	1.63282	236.125
2	79.811	68.8607	14.5356	220.237
3	80.0308	67.4767	9.06303	237.155
4	76.6054	67.1228	11.9046	247.542
5	78.4138	64.7282	11.479	255.919
6	74.6505	63.2186	13.5364	269.068
7	75.6533	60.8888	14.8536	275.206
8	76.263	58.8192	17.275	281.063
9	75.3432	60.0002	20.2097	292.363
10	75.9921	59.18	25.4992	293.228
11	77.0771	58.536	24.8055	298.945
12	77.266	58.8874	29.9044	301.238
13	77.2446	58.6144	38.4335	303.402
14	75.8897	57.6507	40.9974	306.625
15	76.1838	56.9167	47.1452	302.539
16	74.7502	57.2922	48.5507	303.932
17	75.5604	56.2305	49.8083	293.677
18	77.5976	57.6505	52.7721	295.563
19	73.5316	53.9505	53.9362	291.335
20	71.8831	51.5204	54.468	286.074
21	72.4515	53.513	55.9091	284.233
22	72.4088	55.7861	56.5009	275.515
23	71.3812	52.5812	58.0657	276.425
24	64.3445	46.0797	58.2322	246.443

Table B.5: Test System 1, Case2, RCGA-PSO: Average Best Chromosome,Average best fitness = 925355, Average worst fitness = 955162, CPU time =982.41 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$
1	8	6 27.8956	17.1177	1014.36	
2	7.74487	8.58901	25.697	15.4631	1032.16
3	7.29311	8.4255	22.4719	15.837	976.312
4	8.14375	9.14305	23.5933	14.6717	901.791
5	8.97431	8.36195	22.9374	16.7435	879.261
6	8.121	8.13179	21.8311	16.1515	999.205
7	7.70393	9.29346	21.2959	16.4741	1229.86
8	7.58129	8.03989	20.8457	15.3593	1584.24
9	8.66724	8.31101	19.296	16.0394	1805.36
10	9.22517	7.91983	18.3367	17.612	1867.36
11	7.69241	8.43884	18.4225	16.3002	1788.97
12	8.81629	8.55097	15.9125	17.9775	1843.4
13	7.97746	8.57328	15.2941	16.3181	1773.1
14	8.26624	8.25086	15.739	17.2493	1737.61
15	7.28279	8.52435	13.709	17.1197	1666.21
16	7.57626	9.45579	14.605	17.9465	1597.58
17	7.82829	7.7587	14.0437	16.782	1675.39
18	8.93567	7.70525	15.8649	16.9948	1683.22
19	7.15423	8.40786	15.38	17.9823	1783.37
20	8.808	8.5474	14.3317	18.2234	1812.83
21	7.86401	8.85235	14.9584	17.7381	1782.56
22	8.02934	8.83164	14.7123	18.9653	1653.29
23	7.57904	8.06024	13.9689	18.6497	1396.43
24	8.01451	8.49352	13.5852	18.7308	1131.7

Table B.6: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	66.3	49	5.82617	234.514
2	70.3846	63.2154	10.6844	213.04
3	65.7574	62.6052	22.9592	224.089
4	70.5104	66.0213	16.925	223.895
5	75.4159	60.6501	18.1488	243.34
6	70.0522	58.2564	22.9115	246.255
7	67.364	62.1336	21.5608	256.473
8	67.3898	54.8292	25.8801	251.322
9	74.0128	55.9946	29.7619	261.2
10	77.5512	54.652	32.119	275.286
11	70.7753	57.5435	30.5477	265.999
12	77.1949	57.6379	36.3882	279.105
13	73.052	57.2329	43.8371	266.199
14	76.008	56.6984	40.7363	273.645
15	71.1108	58.3189	47.1161	272.188
16	72.5165	62.279	46.9272	277.206
17	74.1193	53.9736	46.121	270.034
18	80.0906	52.8857	42.9274	271.052
19	70.0894	55.6283	46.5386	275.81
20	79.5919	55.7081	48.6536	275.148
21	74.4103	56.8359	49.5935	270.455
22	76.398	58.5321	51.8469	275.933
23	73.1573	53.9397	52.0789	270.76
24	75.047	56.0035	54.7433	272.46

Table B.7: Test System 1, Case2, RCGA-SOHPSO: Average Best Chromosome,Average best fitness = 899068, Average worst fitness = 946753, CPU time =963.69 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$
1	15	6.03922	27.4125	21.4171	856.802
2	7.49966	8.4236	18.3039	16.3545	810.446
3	7.99059	8.40951	19.5764	16.676	864.033
4	7.68803	8.38163	19.0888	16.6289	820.649
5	7.70279	8.37847	19.0003	16.4541	786.029
6	7.87999	8.56911	18.4732	16.7574	840.224
7	7.81305	8.49104	17.6921	16.7126	1137.7
8	7.71602	8.44785	18.047	16.6055	1516.38
9	7.6605	8.34207	16.928	16.78	1765.18
10	7.66912	8.47246	17.1459	16.4055	1874.4
11	7.52193	8.4164	18.0323	16.6568	1786.46
12	7.81134	8.5027	18.051	16.4947	1844.76
13	7.85714	8.28015	17.8642	16.6316	1776.83
14	7.68183	8.33136	17.4993	16.8131	1740.76
15	7.56782	8.34664	17.5402	16.6845	1669.09
16	7.87261	8.41791	16.8136	16.6656	1582.04
17	7.82871	8.68097	17.0213	16.7242	1676.49
18	8.03466	8.45696	16.8116	16.6924	1679.32
19	7.60258	8.35153	17.2909	16.5191	1781.54
20	7.75544	8.67655	16.9836	16.7947	1792.86
21	7.8968	8.46191	17.0605	16.8582	1768.54
22	7.75692	8.42215	17.1272	17.0488	1653.37
23	8.21689	8.59502	17.3739	16.5703	1361.43
24	7.59571	8.31448	17.0725	16.9456	1071.87

Table B.8: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	98.5	49.2653	10.2933	258.78
2	71.7819	64.0914	45.3009	220.978
3	74.6244	64.3417	41.7155	235.841
4	72.7676	64.4691	42.4243	236.928
5	72.0753	64.2071	43.497	237.002
6	72.8785	64.2896	46.1255	239.786
7	72.2743	62.2429	50.23	241.28
8	72.0503	61.1777	49.0487	241.209
9	72.5323	60.3181	52.8184	243.887
10	73.3429	61.3241	52.2222	242.24
11	73.4416	61.3799	49.0992	245.006
12	75.5645	61.4662	48.8188	244.312
13	76.3455	59.9618	49.616	246.857
14	76.0289	60.6598	51.8243	249.488
15	75.7769	61.1701	52.6468	249.931
16	77.962	61.187	55.0048	251.594
17	77.6179	61.5671	54.2513	253.861
18	79.2729	58.7612	54.9335	255.076
19	76.1548	57.3321	53.0983	255.623
20	77.078	58.6936	54.5165	258.699
21	77.8045	57.9532	54.0817	260.087
22	76.7603	58.1806	54.0705	262.492
23	79.8344	58.7432	52.4882	260.725
24	76.2229	57.0158	53.364	264.611

Appendix C

Table for Test System 2, Case 1

Table C.1: Test System 2, Case1, RCGA: Average Best Chromosome,Average best fitness = 44926.8, Average worst fitness = 49811.5, CPU time = 946.33

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$	$P_s(2,t)$	$P_s(3,t)$
1	11.4457	6.3	29.8281	17	108.948	161.872	138.093
2	7.90411	7.89401	17.9337	16.3347	79.7753	155.76	142.592
3	7.83467	8.7975	18.8026	16.1203	66.7125	93.5818	117.536
4	7.95456	8.137	18.6841	15.6137	73.7743	69.0391	89.4371
5	7.70685	8.52737	18.4447	16.9329	67.5078	85.0032	88.0706
6	7.80512	8.23525	17.8995	16.1731	70.6062	142.867	159.347
7	7.993	8.43823	18.0994	15.9354	109.637	208.026	204.085
8	8.02112	8.08024	18.7906	16.6475	128.269	194.582	256.503
9	7.90535	8.66066	17.9437	16.5988	134.826	256.333	261.938
10	8.40289	8.7109	18.3996	16.5758	131.644	245.303	260.723
11	8.01889	8.56133	17.4743	16.7956	135.226	227.08	293.253
12	8.13231	8.75444	17.7688	16.3982	143.025	241.807	319.055
13	8.47342	8.09286	17.1495	16.9532	137.438	210.433	310.432
14	8.25096	8.09474	17.9236	16.9711	139.959	224.644	214.291
15	7.8689	8.69737	18.0079	16.8108	94.2826	188.286	274.976
16	8.03669	8.71759	17.1329	17.3604	138.825	241.225	219.504
17	8.5453	8.53868	17.4691	17.0107	138.393	205.638	245.449
18	7.79056	8.47567	17.3968	17.567	112.732	226.431	324.004
19	8.03784	8.36339	17.5608	17.4678	137.478	237.039	238.697
20	8.65221	8.75875	17.3387	17.6087	132.227	223.693	230.948
21	7.96704	8.30456	17.7798	18.0351	86.3067	149.041	216.753
22	7.82883	8.56465	17.1123	17.7663	102.307	145.971	153.839
23	7.83084	8.6299	16.856	17.6053	71.2624	135.504	185.259
24	7.90613	8.51025	17.2547	17.8531	62.3249	148.186	132.609

Table C.2: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	82.4632	50.32	0.595451	233.17
2	73.6419	60.3926	47.2985	220.54
3	73.5037	65.9262	45.0911	237.649
4	73.982	62.7112	44.7932	236.263
5	71.7904	64.6	45.1692	247.859
6	72.0999	62.138	47.8018	245.14
7	73.3606	61.7921	47.0532	246.046
8	73.9529	58.9214	45.1872	252.585
9	73.9767	62.0512	47.0532	253.822
10	77.6836	62.2579	46.2717	256.117
11	76.0604	61.7746	47.2463	259.361
12	77.6105	62.4747	47.8788	258.149
13	80.4424	58.6244	49.7476	262.883
14	79.4964	59.0587	48.584	263.967
15	77.3185	62.8652	48.9728	263.299
16	78.5136	62.5262	51.4468	267.959
17	82.5306	60.4061	51.1385	266.445
18	77.1596	58.2895	51.0312	270.352
19	78.9949	56.6481	51.5222	269.621
20	82.729	58.4963	51.7104	270.197
21	77.9188	56.3539	50.6031	273.024
22	76.7547	58.1735	52.4372	270.518
23	77.09	58.1554	52.9626	269.767
24	77.6431	56.9576	51.8359	270.444

Table C.3: Test System 2, Case 1, RCGA-AFSA: Average Best Chromosome, Average best fitness = 45842.2, Average worst fitness =49590.3 , CPU time = 1289.15 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$	$P_s(2,t)$	$P_s(3,t)$
1	10.0244	6.46047	29.1024	16.6009	103.41	144.599	158.314
2	8.09369	8.40049	18.9704	16.3804	100.82	142.511	131.892
3	7.47744	8.37809	18.0077	15.8832	63.7483	101.475	117.917
4	8.4416	8.57433	18.1401	16.6455	52.7925	83.5152	83.0062
5	8.4017	8.20928	18.3878	16.3869	71.2228	83.517	87.5736
6	7.83907	8.71443	18.9085	16.6202	89.5729	134.594	148.013
7	7.82912	8.55389	18.5075	16.8467	99.9408	202.949	217.178
8	8.04145	8.09845	18.6709	16.2643	130.997	185.926	266.276
9	8.2967	8.35868	18.0681	16.56	123.208	224.296	306.735
10	8.28673	8.29864	17.8549	16.608	127.245	227.038	285.153
11	8.04001	8.58796	17.7642	17.131	129.198	230.098	294.043
12	7.98625	8.55494	18.3825	16.7634	133.842	236.055	338.198
13	8.31726	8.20385	17.1384	16.7677	137.819	243.988	281.298
14	8.7134	8.30842	17.995	17.1283	115.3	184.738	274.835
15	8.07507	8.38472	17.5076	17.2566	101.32	190.76	262.049
16	7.93858	8.42011	17.6069	17.0381	131.021	224.472	250.847
17	7.77989	8.25659	16.8441	17.9861	137.314	218.576	234.807
18	8.00195	8.27966	17.687	16.352	129.442	244.458	298.813
19	8.48772	8.60087	17.9062	16.9652	141.153	240.691	233.648
20	8.91382	7.93804	16.9509	17.2057	137.049	207.04	249.011
21	8.20554	9.01373	17.1935	17.24	107.397	131.606	211.049
22	8.10495	8.85281	16.8932	17.6666	97.4914	166.053	134.502
23	8.37356	8.51438	16.6647	17.5665	92.0483	145.005	152.169
24	7.89681	8.80868	17.7263	17.3639	68.1109	94.0526	183.883

Table C.4: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	77.8797	51.2585	3.2991	233.844
2	75.1014	63.6216	45.1521	220.88
3	71.5137	63.4921	46.1121	235.418
4	76.7915	64.9496	45.3192	243.137
5	75.7772	62.9004	44.9603	243.38
6	72.2469	64.5153	43.8575	246.625
7	72.3077	62.0304	45.5198	249.739
8	73.8914	59.0192	44.9289	248.533
9	76.0328	60.3538	46.7669	252.234
10	77.0031	60.3988	47.4611	255.328
11	76.4277	62.245	47.1272	260.269
12	76.4366	61.7425	44.6551	258.92
13	78.6179	59.7894	48.4216	259.745
14	82.1521	60.6134	48.1261	263.576
15	78.8403	61.6727	50.4071	264.459
16	77.8208	61.7644	49.6473	263.957
17	77.2009	59.432	52.1412	269.993
18	78.5674	58.1728	50.1349	259.809
19	81.5516	58.6233	49.1196	264.213
20	83.5517	54.669	51.2918	266.738
21	79.3143	60.602	52.0089	267.609
22	78.529	60.1256	53.4712	269.624
23	80.3669	57.8546	53.759	268.412
24	76.8223	58.8692	51.2009	266.762

Table C.5: Test System 2, Case 1, RCGA-PSO: Average Best Chromosome, Average best fitness = 46442.9, Average worst fitness =76115.4, CPU time = 1386.52 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$	$P_s(2,t)$	$P_s(3,t)$
1	8	7.78168	27.3333	15.6333	75.0343	174.535	143.701
2	7.87823	9.22813	22.8852	15.1298	70.0558	161.47	183.773
3	8.58549	8.95626	23.3717	14.6456	56.3009	127.48	140.8
4	8.7438	9.0287	21.5569	16.7892	72.0873	84.5381	92.6743
5	8.32215	8.35839	17.9549	14.4723	66.5705	104.827	99.5112
6	8.67683	8.59605	18.8829	17.9237	59.6061	154.813	161.879
7	7.53444	8.03295	19.7841	15.7466	97.2736	161.14	290.436
8	8.05699	7.66281	20.7327	15.4783	141.116	222.27	244.68
9	8.85924	8.40433	20.2599	15.1087	143.542	237.048	298.178
10	6.97521	9.49623	20.1282	16.8017	129.377	229.816	301.107
11	8.11212	8.98305	19.0777	16.7866	128.464	215.798	322.249
12	8.58942	8.56465	19.0169	17.7163	146.891	229.507	333.428
13	8.60137	7.71711	16.162	16.1849	157.858	243.323	268.092
14	9.15951	7.99832	16.1092	17.1724	133.843	216.693	224.713
15	7.03704	7.30957	17.9253	16.5199	108.001	195.625	274.532
16	6.56093	8.63408	15.9116	17.404	121.231	216.648	271.7
17	7.84144	8.69173	16.9571	17.4413	129.772	202.644	267.226
18	8.43938	8.30328	15.112	17.8539	151.351	237.346	272.406
19	9.08738	9.53825	15.4361	16.5295	142.194	229.891	242.629
20	8.34218	7.56753	13.2188	17.1093	115.247	201.699	280.221
21	7.20089	7.9528	13.7471	19.9524	79.0021	174.234	190.391
22	7.58441	7.2762	12.5659	18.0606	87.3726	166.944	154.317
23	8.6046	9.21585	15.8342	18.4442	92.6836	119.78	184.819
24	6.54305	9.02749	16.1129	19.2526	73.4228	147.688	139.216

Table C.6: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	66.3	56.8858	7.47467	226.069
2	67.9093	65.2664	22.0809	208.897
3	72.763	63.1737	16.0994	212.68
4	71.5036	63.365	22.4453	234.419
5	67.8013	59.7005	32.4668	228.247
6	70.1221	60.1183	27.9572	254.281
7	63.7929	54.936	26.4589	243.577
8	67.1626	51.7146	24.8151	246.007
9	72.4499	54.8385	25.5134	247.876
10	62.5792	59.9542	24.1419	261.25
11	70.214	56.8552	29.4078	261.202
12	72.3209	54.3726	28.0661	270.096
13	74.9554	51.4852	35.3434	264.906
14	78.3163	52.6033	37.4766	273.137
15	66.508	50.3342	32.7982	269.712
16	64.8627	56.7502	41.1484	277.415
17	71.1573	55.1228	35.4563	276.478
18	75.2047	51.5617	43.2147	279.515
19	78.6413	56.5614	41.7482	269.546
20	74.7742	47.9583	49.1584	269.119
21	68.4879	50.2327	50.5569	287.077
22	70.4828	47.6365	52.789	271.311
23	75.6723	58.0135	43.7166	271.37
24	64.2934	56.7472	45.8238	272.684

Table C.7: Test System 2, Case 1, RCGA-SOHPSO: Average Best Chromosome, Average best fitness = 46320.8, Average worst fitness =54066.6, CPU time = 1347.1 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$	$P_s(2,t)$	$P_s(3,t)$
1	15	6	30	21.4	90.1074	121.21	133.989
2	8.13014	8.34386	19.2308	17.1417	71.4612	139.126	164.296
3	7.93354	8.4744	19.1933	17.1289	67.4057	92.7776	116.284
4	8.53775	8.43451	18.3175	16.8603	51.2072	83.762	72.2349
5	8.49814	8.6132	18.0735	17.3134	60.0699	86.6789	86.7014
6	8.08302	8.34101	18.9111	17.0131	80.8344	129.32	160.732
7	8.26532	8.27596	17.5799	16.9324	113.686	198.249	203.483
8	8.38191	8.582	18.6426	17.0803	118.912	212.979	243.68
9	7.94051	8.30075	18.6549	17.0772	116.728	226.726	314.683
10	8.11668	8.53016	18.3996	16.7891	121.503	209.028	313.006
11	8.11728	8.3734	18.0384	17.0071	118.753	216.808	322.718
12	8.36318	8.60847	17.5773	16.8763	107.584	239.016	355.787
13	7.62718	8.44618	17.5774	16.8077	127.067	236.786	301.986
14	7.6715	8.52748	16.7331	17.1039	107.069	218.137	251.758
15	7.76669	8.19037	17.6531	17.1952	116.471	200.476	241.807
16	7.63296	8.56867	17.6251	17.0067	116.38	229.975	262.866
17	8.25478	8.44329	18.08	16.7539	126.285	219.577	253.823
18	7.74596	8.39403	17.9472	16.8348	122.34	224.634	325.736
19	8.13024	8.35283	17.1762	17.023	94.9552	215.932	304.8
20	7.61432	8.38261	17.1735	16.9838	125.946	218.389	254.886
21	7.95519	8.42869	17.2098	16.9654	84.7907	169.532	201.405
22	8.08931	8.36182	17.3749	16.8742	82.4511	133.491	189.941
23	7.81772	8.40932	17.4645	16.8969	66.8209	171.229	159.189
24	8.22217	8.79109	17.936	17.2197	85.2095	111.854	145.636

Table C.8: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	98.5	49	0	257.194
2	75.0709	63.6196	40.6957	225.728
3	74.4684	64.4865	40.8407	241.699
4	76.974	64.6063	44.211	243.064
5	75.7476	65.2985	45.7965	248.695
6	73.7135	62.8198	42.6355	248.943
7	74.2908	61.0249	48.7916	250.03
8	75.128	61.5051	44.5183	252.971
9	74.1513	59.7472	43.8925	254.518
10	75.8505	61.1926	44.4636	254.444
11	77.0038	60.6747	45.8719	257.803
12	78.7279	61.623	48.6634	259.073
13	75.2653	60.4659	48.7402	260.38
14	76.2043	61.4159	53.1094	262.573
15	77.2977	60.0198	50.3028	263.755
16	76.6833	61.7567	50.3681	262.795
17	80.5674	60.1314	48.3801	261.765
18	77.4995	58.1792	49.1658	263.624
19	79.8984	56.9817	52.6106	266.322
20	76.3416	56.8221	52.016	266.292
21	78.3244	57.6502	52.9737	265.643
22	78.8879	57.6413	52.467	265.23
23	77.5533	57.6467	52.3531	265.031
24	80.1392	59.3356	50.8189	266.816

Appendix D

Table for Test System 2, Case 2

Table D.1: Test System 2, Case 2, RCGA: Average Best Chromosome, fitness = 45818.6, Average Worst Fitness = 49850.9, CPU time = 1424.73 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$	$P_s(2,t)$	$P_s(3,t)$
1	11.5323	6.21446	29.4525	16.1825	102.974	143.602	160.258
2	8.41296	8.24201	18.4827	16.1096	92.6208	132.073	149.394
3	8.29117	8.32117	18.8361	15.9294	71.485	112.15	96.6999
4	8.33526	8.06251	18.7412	16.7732	56.6586	77.7066	89.7884
5	8.3266	7.99477	18.2908	16.6098	66.3599	85.8563	89.5179
6	8.41331	8.20823	18.6404	16.9882	81.2981	130.857	155.287
7	8.1992	8.02586	18.2845	16.4249	109.133	212.223	199.784
8	8.38459	8.34836	19.1878	16.6715	140.325	207.862	231.506
9	8.42669	8.79393	17.8688	16.4492	127.035	232.979	292.414
10	8.01717	8.83323	17.8967	16.4672	130.625	237.32	271.606
11	7.6571	7.99067	17.6906	16.6111	133.735	246.663	281.584
12	7.76192	8.22382	17.4753	16.8506	117.362	243.78	344.715
13	7.67977	8.40779	18.0057	16.561	130.626	250.188	285.64
14	8.1814	8.61707	17.7352	16.7655	123.697	191.007	264.045
15	7.58034	8.53472	17.5114	16.5016	112.755	203.456	244.455
16	8.20945	8.47463	18.0405	17.4295	136.794	227.557	236.972
17	8.18341	8.5728	17.1723	17.4825	136.572	215.756	236.741
18	7.7891	8.54618	17.9598	17.3943	115.945	245.43	303.304
19	8.17596	8.94761	17.5989	17.2628	132.527	236.852	241.175
20	8.2937	9.13508	16.9757	18.1679	142.191	204.453	237.061
21	7.84104	8.54743	16.7648	17.9065	93.4368	133.819	222.519
22	7.73931	8.71183	16.3543	16.9091	91.3209	147.819	166.968
23	8.19447	8.54283	17.1654	17.1426	76.3757	163.814	154.601
24	7.65975	8.25817	17.5514	17.6616	67.7121	85.0543	194.933

Table D.2: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	83.556	50.1087	1.47733	231.558
2	77.2081	62.5918	46.9095	219.533
3	76.2789	63.236	43.9856	235.895
4	75.9552	62.2585	43.8528	243.493
5	74.9419	61.9769	46.1795	244.973
6	74.5186	62.4997	45.2831	250.332
7	73.7851	60.4055	46.313	248.297
8	74.7983	60.7447	42.7801	252.235
9	75.5166	63.1225	46.916	252.319
10	74.2449	63.6119	47.4689	255.123
11	72.8763	59.3871	47.6182	257.444
12	73.9441	60.6813	48.7527	260.534
13	74.4148	61.2576	47.9453	260.031
14	77.967	62.8401	48.5806	262.263
15	74.9801	62.9501	49.509	262.211
16	78.9658	62.2138	48.9042	269.325
17	79.0747	61.9122	51.0688	270.031
18	76.7405	59.9426	50.0151	269.749
19	79.582	60.6159	50.7965	269.369
20	79.6634	60.8273	52.2138	274.651
21	76.5176	58.1576	53.2336	273.652
22	75.812	58.9385	54.1513	265.796
23	78.6282	57.6555	52.3853	266.819
24	75.7822	55.6906	51.7649	268.917

Table D.3: Test System 2, Case 2, RCGA-AFSA: Average Best Chromosome, Average Best Fitness = 44490.7, Average Worst Fitness = 49972.7, CPU time = 1504.63 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$	$P_s(2,t)$	$P_s(3,t)$
1	11.1578	6.10568	29.8395	15.6116	100.573	131.819	151.631
2	9.54391	9.94507	24.2734	17.5862	97.6488	130.987	154.881
3	9.66719	9.65546	25.0772	17.7016	82.2063	112.881	100.466
4	9.24086	9.81368	23.7331	17.505	70.8926	73.512	91.2071
5	9.31762	9.65865	22.9113	17.6583	73.8577	80.8359	90.62
6	9.16517	9.68355	23.249	18.2959	85.5453	139.383	145.691
7	8.88697	9.19726	24.0965	17.7358	96.0262	197.079	238.752
8	9.10419	9.07516	22.2394	18.4731	104.935	186.113	282.123
9	8.90746	9.11911	21.9183	18.8344	108.483	216.258	320.675
10	8.7863	8.51222	21.6031	18.2964	119.09	214.14	304.199
11	8.55396	8.66965	20.8889	18.266	111.317	209.927	327.753
12	8.26111	8.4424	20.0422	18.3084	120.745	222.834	351.83
13	7.88924	8.42484	18.1624	18.4169	117.486	212.426	315.401
14	7.97298	8.20928	16.834	19.0363	110.136	203.626	238.485
15	7.91164	8.17142	16.2161	18.4022	106.753	174.635	252.919
16	7.22624	7.86717	14.8567	17.4976	104.316	198.419	290.987
17	7.13692	7.74821	14.0414	17.7022	108.27	204.636	269.039
18	7.53327	7.8258	12.79	17.9255	111.392	207.848	330.743
19	7.09963	7.26626	12.8086	16.7372	114.654	208.403	293.816
20	6.99215	7.49994	12.4466	16.9858	104.552	212.629	281.589
21	6.71626	7.51126	11.5452	16.8671	100.752	174.655	186.055
22	6.67189	7.50275	11.9056	16.5305	97.2929	168.483	150.083
23	6.78017	7.50694	12.2939	15.8258	91.5564	162.734	159.699
24	6.32805	7.00556	12.1111	14.9559	106.044	119.748	152.611

Table D.4: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	86.7902	49.6582	0	230.699
2	83.021	70.7425	16.188	227.692
3	82.9861	68.9586	7.71667	243.53
4	79.9904	68.8487	13.6682	250.211
5	79.0871	67.1019	16.6782	260.108
6	77.5342	65.4408	14.7098	270.529
7	75.7132	60.5595	9.20273	272.337
8	76.5478	58.6605	19.402	282.024
9	75.9492	57.7902	20.6227	290.272
10	76.1485	54.9899	20.9984	290.371
11	76.4108	56.069	24.838	294.101
12	75.1982	54.6331	28.4377	296.946
13	73.7498	54.2888	37.0968	299.971
14	75.5156	53.6553	42.6766	305.806
15	75.852	53.8364	45.0229	300.993
16	71.8893	52.4238	48.9312	293.467
17	71.6612	51.2479	51.457	294.27
18	74.1977	50.2829	53.3006	293.291
19	71.2704	47.0446	54.3155	281.816
20	70.1877	48.7297	54.2202	279.305
21	68.283	50.0265	56.2564	274.717
22	68.2551	50.9438	57.1857	268.426
23	69.1613	51.1754	57.8334	258.468
24	66.216	49.0485	58.3339	248.573

Table D.5: Test System 2, Case 2, RCGA-PSO: Average Best Chromosome, fitness = 46455.1, CPU time = 1424.73 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$	$P_s(2,t)$	$P_s(3,t)$
1	8	7.5	28.6667	14.9832	43.6974	138.929	185.38
2	8.65524	7.46251	22.5961	16.7413	62.0793	156.483	193.811
3	8.85997	8.61181	23.8335	15.5529	69.7742	139.956	110.406
4	8.5094	8.17508	21.0988	16.6389	59.1376	81.8255	115.099
5	7.91233	8.8864	20.1832	15.6379	63.5828	86.1222	120.524
6	7.45945	9.70757	22.4679	14.4675	87.269	142.419	152.909
7	8.20562	8.27586	20.456	17.0181	102.105	162.639	274.762
8	7.14861	9.57684	21.5276	16.647	105.997	154.788	336.336
9	7.21959	9.15305	22.0535	16.8982	142.133	190.894	326.888
10	9.16043	8.25896	19.5251	17.6159	145.535	234.236	265.996
11	8.11345	9.1739	16.0671	16.3669	144.68	173.814	343.406
12	8.84066	9.10298	17.7637	17.0555	139.447	200.073	366.999
13	8.80582	9.54809	19.1409	16.1331	142.934	229.865	301.13
14	8.03321	8.35018	16.8289	18.2888	115.067	184.971	280.68
15	9.16441	8.04488	17.4835	18.7886	123.83	197.824	229.362
16	7.34945	8.65538	14.7661	16.7318	123.713	217.438	272.138
17	8.09308	8.25759	15.2944	16.6096	133.836	235.825	233.185
18	7.58036	7.4357	14.3683	18.9197	139.022	239.914	286.436
19	8.9273	8.7953	15.8417	19.0764	138.384	198.426	271.252
20	7.52099	8.60853	14.2492	18.7883	123.317	183.711	287.287
21	7.69978	8.07013	14.9764	19.0484	96.9409	161.723	202.214
22	7.78997	7.26088	13.6126	17.3791	91.2877	131.962	185.802
23	7.42966	7.29894	14.9962	17.625	95	165.461	159.457
24	6.84993	7.81924	13.2828	19.1876	68.9765	110.07	159.706

Table D.6: Average Best P_h

t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	66.3	55.6	3.73733	222.357
2	71.8632	55.4336	21.6275	217.297
3	73.7762	60.8014	17.6049	222.737
4	71.0814	58.9491	23.289	235.001
5	68.5309	62.4697	26.3443	232.574
6	64.4368	65.9011	19.3716	228.25
7	68.6992	57.3973	26.3694	245.026
8	64.8375	62.3567	22.5879	248.406
9	65.4632	59.3976	20.1528	253.692
10	75.7264	55.797	27.9637	261.637
11	71.904	60.5407	35.4424	258.077
12	75.1298	59.6152	33.6711	263.381
13	77.235	60.9333	29.5555	256.642
14	73.0586	56.3553	35.8541	271.203
15	79.2745	55.5748	34.0577	278.602
16	70.2484	58.3787	43.314	264.491
17	73.4566	55.4835	40.7476	265.982
18	70.4904	50.1921	45.8352	279.609
19	77.5523	55.8084	42.4212	281.629
20	69.7456	54.3025	46.6044	277.195
21	70.6855	51.7938	45.3344	278.672
22	71.7376	49.0092	50.6431	266.109
23	69.6682	49.3829	46.0202	263.915
24	67.6479	51.2408	53.1359	273.326

Table D.7: Test System 2, Case 2, RCGA-SOHPSO: Average Best Chromosome, Average Best Fitness = 46333.8, Average Worst Fitness = 51146.1, CPU time = 1424.73 secs

t	$Q_h(1,t)$	$Q_h(2,t)$	$Q_h(3,t)$	$Q_h(4,t)$	$P_s(1,t)$	$P_s(2,t)$	$P_s(3,t)$
1	13.6667	7.8	30	17.6336	40.0071	128.543	166.506
2	9.17013	9.1095	23.2195	16.8992	85.4577	128.023	180.231
3	8.67381	9.13668	24.3252	17.9807	54.7446	128.47	115.66
4	8.65556	9.35013	22.7799	18.5847	72.9377	89.3652	72.5595
5	8.99595	9.00491	23.6349	17.8673	61.1275	108.186	88.4147
6	8.96082	9.9667	21.9404	17.7651	80.3778	133.997	161.605
7	8.26869	9.56621	23.4195	18.1747	96.2822	173.976	260.429
8	8.57261	8.61838	19.7983	18.2879	114.665	204.292	247.158
9	9.0034	9.46889	21.2373	18.3422	146.791	265.495	241.82
10	8.23807	9.16824	20.7292	18.2923	108.119	207.946	327.503
11	8.32418	9.64764	21.1741	18.2223	128.429	208.913	322.818
12	8.42468	8.70107	20.8506	19.0409	129.235	226.339	348.452
13	8.98503	8.54957	19.0528	17.9848	132.793	227.22	302.927
14	8.0673	8.47705	18.4137	18.1941	124.445	231.026	223.904
15	7.4305	8.42262	15.3965	17.4452	109.718	192.772	249.888
16	8.01082	8.66884	14.7657	18.2383	114.776	173.636	303.442
17	7.62139	7.67202	15.4825	17.5335	116.919	200.225	280.385
18	6.9127	7.10627	15.1218	18.1461	127.442	230.298	318.163
19	7.42553	7.78595	13.9382	17.1844	110.703	196.788	315.106
20	7.32485	7.25098	12.7821	16.9978	103.097	225.191	274.284
21	7.20084	7.1775	12.676	17.1546	112.358	190.684	160.3
22	6.86264	7.08682	12.3113	16.4726	89.7283	173.121	157.895
23	6.66137	7.09405	12.9805	15.9605	76.199	167.789	174.559
24	7.11316	7.32087	12.9542	16.5023	68.5022	136.607	160.056

Table D.8: Average Best P_h

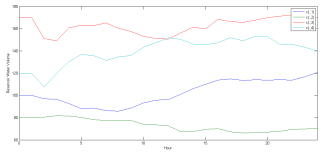
t	$P_h(1,t)$	$P_h(2,t)$	$P_h(3,t)$	$P_h(4,t)$
1	92.3667	56.92	0	236.991
2	79.0738	64.0645	20.073	217.654
3	75.3142	62.5805	13.1145	231.159
4	73.8653	62.4705	17.1195	238.384
5	73.5094	58.9528	13.9986	239.318
6	71.9344	60.3774	20.7621	243.624
7	67.7671	54.8883	16.577	249.017
8	68.9365	48.273	27.6403	254.18
9	70.3059	51.2515	24.115	258.054
10	67.3103	50.2291	25.9037	262.195
11	69.683	52.1741	24.5672	267.206
12	70.9374	47.3865	26.9377	275.319
13	74.4563	45.655	32.1389	269.399
14	71.1293	46.1168	32.9223	274.346
15	68.9609	46.9393	45.4886	273.27
16	72.435	48.9262	46.6476	280.006
17	72.1549	44.0942	45.5794	275.875
18	68.0164	40.1854	45.9338	279.14
19	70.9707	43.7011	50.2882	273.398
20	71.1948	42.8635	54.513	271.175
21	70.1766	44.3247	55.2213	271.208
22	68.9617	46.0173	56.9828	263.59
23	68.0915	47.6176	56.794	257.463
24	71.5714	50.2666	55.8361	258.078

Appendix E

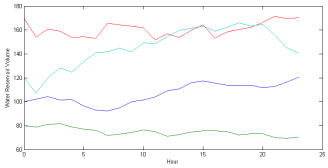
Reservoir Volume Violations

system	case	method	Reservoir 1	Reservoir 2	Reservoir 3	Reservoir 4	Total
1	1	RCGA	0.0005%	0.0027%	0.0035%	0.0013%	0.008%
1	1	RCGA-AFSA	0.0020%	0.0032%	0.0006%	0.0018%	0.0076%
1	1	RCGA-PSO	0.0030%	0.0116%	0.00004%	0.0035%	0.01814%
1	1	RCGA-SOHPSO	0.0069%	0.0028%	0.0035%	0.0029%	0.0161%
1	2	RCGA	0.0052%	0.0043%	0.0015%	0.0054%	0.0164%
1	2	RCGA-AFSA	0.0021%	0.0032%	0.0004%	0.0026%	0.0083%
1	2	RCGA-PSO	0.0081%	0.0080%	0.0009%	0.0014%	0.0185%
1	2	RCGA-SOHPSO	0.0009%	0.0022%	0.0006%	0.0004%	0.0041%
2	1	RCGA	0.0015%	0.0133%	0.0082%	0.0004%	0.027%
2	1	RCGA-AFSA	0.0038%	0.0041%	0.0020%	0.0020%	0.0119%
2	1	RCGA-PSO	0.0042%	0.0088%	0.0032%	0.0100%	0.0262%
2	1	RCGA-SOHPSO	0.0008%	0.0023%	0.0006%	0.0028%	0.0065%
2	2	RCGA	0.0005%	0.0118%	0.0051%	0.0005%	0.0179%
2	2	RCGA-PSO	0.0029%	0.0087%	0.0033%	0.0093%	0.0242%
2	2	RCGA-SOHPSO	0.0038%	0.0101%	0.0004%	0.0055%	0.0198%

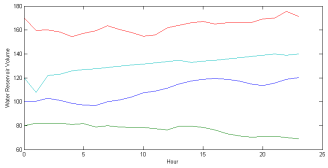
E.1 RCGA



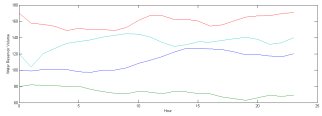
(a) Test System 1, Case 1



(b) Test System 1, Case 2



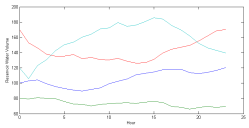
(c) Test System 2, Case 1



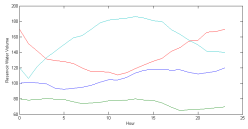
(d) Test System 2, Case 2

Figure E.1: Hourly reservoir storage volumes obtained by RCGA

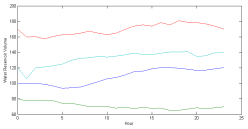
E.2 RCGA-AFSA



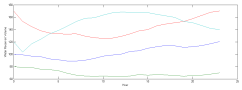
(a) Test System 1, Case 1



(b) Test System 1, Case 2



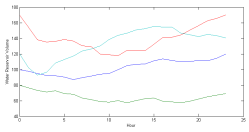
(c) Test System 2, Case 1



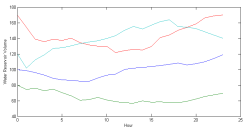
(d) Test System 2, Case 2

Figure E.2: Hourly reservoir storage volumes obtained by RCGA-AFSA

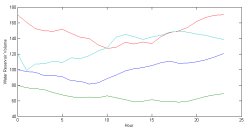
E.3 RCGA-PSO



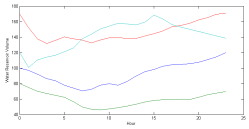
(a) Test System 1, Case 1



(b) Test System 1, Case 2



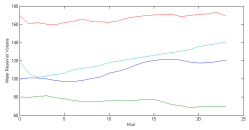
(c) Test System 2, Case 1



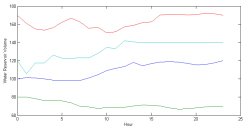
(d) Test System 2, Case 2

Figure E.3: Hourly reservoir storage volumes obtained by RCGA-PSO

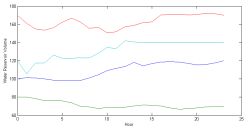
E.4 RCGA-SOHPSO



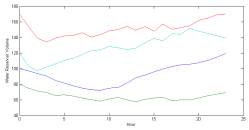
(a) Test System 1, Case 1



(b) Test System 1, Case 2



(c) Test System 2, Case 1



(d) Test System 2, Case 2

Figure E.4: Hourly reservoir storage volumes obtained by RCGA-SOHPSO

Appendix F

Power Generation Violations

system	case	method	Violation
1	1	RCGA	0 %
1	1	RCGA-AFSA	1.870%
1	1	RCGA-PSO	0.560%
1	1	RCGA-SOHPSO	2.265%
1	2	RCGA	0%
1	2	RCGA-AFSA	0.188%
1	2	RCGA-PSO	0.532%
1	2	RCGA-SOHPSO	2.640%
2	1	RCGA	0.11%
2	1	RCGA-AFSA	0.054%
2	1	RCGA-PSO	1.04%
2	1	RCGA-SOHPSO	0.056%
2	2	RCGA	0.136%
2	2	RCGA-AFSA	0.020%
2	2	RCGA-PSO	1.338%
2	2	RCGA-SOHPSO	2.064%

Appendix G

Source Code for Test System 1, all cases

```
#include <iostream>
#include <deque>
#include <cstdlib>
#include <ctime>
#include <cmath>
#define POP_SIZE 100
//Constants
double T=24; //time interval
double iterators = 30; //iterations?
double N_h = 4; //number of hydro plants
double N_s = 1; //number of thermal plants
int N_p = 100; //total number of population
int P_e = 10; //total number of elite individuals in N_p
int P_c = 40; //population from crossover
int P_m = 50; //population number of mutated individuals
double step = .5; //step for AFSA
double visual = 1; //visual for AFSA
double e_vcourse = 1, e_vfine = .01;//for constraints handling
double e_pcourse = 2, e_pfine = .01;
double iteration_course = 20, iteration_fine = 10;
double P_smin = 500;
double P_smax = 2500;
double Q_hmin[4] = {5,6,10,13};
double Q_hmax[4] = {15,15,30,25};
double V_hmin[4] = {80,60,100,170};
double V_hmax[4] = {150,120,240,160};
double V_hbegin[4] = {100,80,170,120};
double V_hend[4] = {120,70,170,140};
double a_si = 5000, b_si = 19.2, c_si = .002, d_si = 700, e_si = .085;
double R_u[4] = {0,0,2,1};
double P_D[24] = {1370, 1390, 1360, 1290, 1290, 1410, 1650, 2000, 2240, 2320, 2230, 2310, 2230, 2200, 2130, 2070, 2130, 2140, 2240, 2280, 2240, 2120, 1850, 1590};
double time_delay[4] = {2,3,4,0};
double gmax_rcga = 1000;
//change the iteration number depending on the hybrid. 0 to disable method
double gmax_afsa = 400;
double gmax_pso = 400;
double b_mutation = 1;
double n_c = 1;
double c1[4] = {-0.0042, -0.0040, -0.0016, -0.0030};
double c2[4] = {-0.42, -0.30, -0.30, -0.31};
double c3[4] = {0.030, .015, 0.014, 0.027};
double c4[4] = {0.90, 1.14, 0.55, 1.44};
double c5[4] = {10, 9.5, 5.5, 14};
double c6[4] = {-50, -70, -40, -90};
double inflow[4][24];
double crowd_factor = .8;
int upstreamIndex[4][4];

using namespace std;

deque<deque<deque<double> > > population;// the last element in 3d deque is the fitness value
deque<double> population_fitness;
```

```
deque<deque<deque<double> > > volume;
deque<deque<deque<double> > > population_elite;
deque<deque<deque<double> > > population_crossover;
deque<deque<deque<double> > > population_mutation;
int board[POP_SIZE][POP_SIZE];//map, list of visible fishes
deque<deque<double> > populationS;
deque<deque<double > > populationF;
double S_fitness=0;
double F_fitness=0;
double bulletin;
double best_fitness;
deque<double> aveBestFitness;
deque<double> aveWorstFitness;
deque<deque<double> > aveBestChromosome;
deque<deque<double> > aveWorstChromosome;
deque<deque<double> > aveBestPh;
deque<deque<double> > aveBestPs;
deque<deque<double> > aveBestQh;
double aveTime;
//FUNCTIONS
void inflowF(){//[37]
    double huehue1[24] = {10, 9, 8, 7, 6, 7, 8, 9, 10, 11, 12, 10, 11, 12, 11, 10, 9, 8, 7, 6, 7, 8, 9, 10};
    double huehue2[24] = {8, 8, 9, 9, 8, 7, 6, 7, 8, 9, 9, 8, 8, 9, 9, 8, 7, 6, 7, 8, 9, 9, 8, 8};
    double huehue3[24] = {8.1, 8.2, 4, 2, 3, 4, 3, 2, 1, 1, 1, 2, 4, 3, 3, 2, 2, 2, 1, 1, 2, 2, 1, 0};
    double huehue4[24] = {2.8, 2.4, 1.6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

    for(int i=0;i<24;i++){
        inflow[0][i] = huehue1[i];
    }
    for(int i=0;i<24;i++){
        inflow[1][i] = huehue2[i];
    }
    for(int i=0;i<24;i++){
        inflow[2][i] = huehue3[i];
    }
    for(int i=0;i<24;i++){
        inflow[3][i] = huehue4[i];
    }
}

void initUpstream(){//initialize what plant is upstream of the jth hydro plant, if -1, no plant upstream
    for(int i=0;i<N_h;i++){
        for(int j=0;j<N_h;j++){
            upstreamIndex[i][j] = -1;
        }
    }
    upstreamIndex[2][0] = 0;
    upstreamIndex[2][1] = 1;
    upstreamIndex[3][0] = 2;
}

double sum_upstream(int pop,int t,int j){
    double temp=0;
    for(int i=0;i<R_u[j];i++){
        if((t-time_delay[i])<0){
            continue;
        }
        else{
            if(isinf(population[pop][t-time_delay[i]][j]) || isinf(population[pop][t-time_delay[i]][j])!=isinf(population[pop][t-time_delay[i]][j])){
                population[pop][t-time_delay[i]][j] = Q_hmax[j];
                continue;
            }
            else{//////////////////////////////////////////
```

```
                    temp = temp + population[pop][t-time_delay[i]][upstreamIndex[j][i]];
            }
        }
    }
    return temp;
}

void check_discharge(int index,int j){//check every Q_h for all t in jth hydro plant
    for(int t=0;t<T;t++){
        if(population[index][t][j]<Q_hmin[j]){
            population[index][t][j] = Q_hmin[j];
            if(population[index][t][j]!=population[index][t][j] || isinf(population[index][t][j])){
                population[index][t][j] = Q_hmax[j];
            }
        }
        if(population[index][t][j]>Q_hmax[j]){
            population[index][t][j] = Q_hmax[j];
            if(population[index][t][j]!=population[index][t][j] || isinf(population[index][t][j])){
                population[index][t][j] = Q_hmax[j];
            }
        }
    }
}

void check_discharge1(int index,int j, int t){//check every Q_h for all t in jth hydro plant
        if(population[index][t][j]<Q_hmin[j]){
            population[index][t][j] = Q_hmin[j];
            if(population[index][t][j]!=population[index][t][j] || isinf(population[index][t][j])){
                population[index][t][j] = Q_hmax[j];
            }
        }
        if(population[index][t][j]>Q_hmax[j]){
            population[index][t][j] = Q_hmax[j];
            if(population[index][t][j]!=population[index][t][j] || isinf(population[index][t][j])){
                population[index][t][j] = Q_hmax[j];
            }
        }
}

void check_thermal1(int index, int t, int j){
    if(population[index][t][j] <=P_smin){
        population[index][t][j] = P_smin;
        if(population[index][t][j]!=population[index][t][j]){
            }
    }
    if(population[index][t][j] >= P_smax){
        population[index][t][j] = P_smax;
        if(population[index][t][j]!=population[index][t][j]){
            }
    }
}

deque<double> Q_hcF(int index,int j){
    deque<double> temp;
    for(int t=0;t<T;t++){
        if(t==0){
            temp.push_back(population[index][t][j] - Q_hmin[j]);
        }
        else{
            temp.push_back((population[index][t][j]-population[index][t-1][j]));
            if(population[index][t][j]!=population[index][t][j] || population[index][t-1][j]!=population[index][t-1][j] || isinf(population[index][t][j]) || isinf(po
                population[index][t][j] = Q_hmax[j];
                population[index][t-1][j] = Q_hmax[j];
```

```
            }
        }
    }
    return temp;
}

int max_adjustableQ(deque<double> input){
    int itr=0;
    double temp=99999999999999999;
    for(int i=1;i<T;i++){
        if(temp<input[i]){
            temp = input[i];
            itr=i;
        }
    }
    return itr;
}

int max_adjustableP(int index, int t){
    return 0;
}

double P_ssum(int index, int t){
    double temp=0;
    for(int a = N_h; a<N_h+N_s;a++){
        if(population[index][t][a]<0){
            continue;
        }
        temp = temp + population[index][t][a];
    }
    if(temp<0){
        temp =0;
    }
    return temp;
}

double P_hsum(int index, int t){
    double temp=0;
    for(int a = 0; a<N_h;a++){
        double temp1 = c1[a]*pow(volume[index][t][a],2) + c2[a]*pow(population[index][t][a],2) + c3[a]*volume[index][t][a]*population[index][t][a] +
        c4[a]*volume[index][t][a] + c5[a]*population[index][t][a] + c6[a];
        if(temp1<0){
            continue;
        }
        temp = temp + temp1;
    }
    if(temp<0){
        temp = 0;
    }
    return temp;
}

void update_volume(){
    for(int i=0;i<POP_SIZE;i++){
        deque<deque<double> > V_h;//V_h(t,j)
        V_h.clear();
        for(int t=0;t<T;t++){
            deque<double> temp1;
            for(int j=0;j<N_h;j++){
                double temp=0;//V_h(j)
                if((t-1)<0){
                    temp = V_hbegin[j];// + inflow[j][t] - population[i][t][j] + sum_upstream(i,t,j);//spillage not calculated
                    temp1.push_back(temp);
```

```
                }
                else{
                    temp = V_h[t-1][j] + inflow[j][t] - population[i][t][j] + sum_upstream(i,t,j);//spillage not calculated
                    temp1.push_back(temp);
                }
            }
            V_h.push_back(temp1);
            temp1.clear();
        }
        volume[i].swap(V_h);
        V_h.clear();
    }

}

void update_volume(int index){
    deque<deque<double> > V_h;//V_h(t,j)
        V_h.clear();
        for(int t=0;t<T;t++){
            deque<double> temp1;
            for(int j=0;j<N_h;j++){
                double temp;//V_h(j)
                if((t-1)<0){
                    temp = V_hbegin[j];// + inflow[j][t] - population[index][t][j] + sum_upstream(index,t,j);//spillage not calculated
                    temp1.push_back(temp);
                }
                else{
                    temp = V_h[t-1][j] + inflow[j][t] - population[index][t][j] + sum_upstream(index,t,j);//spillage not calculated
                    temp1.push_back(temp);
                }
            }
            V_h.push_back(temp1);
            temp1.clear();
        }
        volume[index].swap(V_h);
        V_h.clear();
}

void evaluation(){//calculate fitness value for population
    population_fitness.clear();
    for(int itr=0;itr<POP_SIZE;itr++){
        double temp=0;
        for(int t=0;t<T;t++){
            for(int i=0;i<N_s;i++){//uncomment for test system 1, case 2
                temp = temp + a_si + b_si*population[itr][t][N_h+i] + c_si*pow(population[itr][t][N_h+i],2);
                // + fabs(d_si*sin(e_si*(P_smin - population[itr][t][N_h+i])));
            }
        }
        population_fitness.push_back(temp);//default place for the fitness value
    }

    //bubble sort
    for(int itr1=0;itr1<POP_SIZE;itr1++){
        for(int itr2=0;itr2<POP_SIZE;itr2++){
            if(population_fitness[itr1]<population_fitness[itr2]){
                population[itr1].swap(population[itr2]);
                double huehue = population_fitness[itr1];
                population_fitness[itr1] = population_fitness[itr2];
                population_fitness[itr2] = huehue;
            }
        }
    }
}
```

```
double evaluationS(){//calculate fitness value for populationF
        double temp=0;
        for(int t=0;t<T;t++){
            for(int i=0;i<N_s;i++){//uncomment the comment for next line for test system 1, case 2
                temp = temp + a_si + b_si*populationS[t][N_h+i] + c_si*pow(populationS[t][N_h+i],2);// + fabs(d_si*sin(e_si*(P_smin - populationS[t][N_h+i])));
            }
        }
        S_fitness = temp;//default place for the fitness value
        return S_fitness;

}

double evaluationF(){//calculate fitness value for populationF
        double temp=0;
        for(int t=0;t<T;t++){
            for(int i=0;i<N_s;i++){//uncomment the comment for next line for test system 1, case 2
                temp = temp + a_si + b_si*populationF[t][N_h+i] + c_si*pow(populationF[t][N_h+i],2);// + fabs(d_si*sin(e_si*(P_smin - populationF[t][N_h+i])));
            }
        }
        F_fitness = temp;//default place for the fitness value
        return F_fitness;

}

void elitist(){//get all elite population and best individual
    population_elite.clear();
    if(best_fitness>population_fitness[0]){
        best_fitness = population_fitness[0];
    }
    for(int i=0;i<P_e;i++){
        population_elite.push_back(population[i]);
    }
}

void sbx(){
    double u;
    do{
        u = (double)rand()/rand();
        u-=(int)u;
    }while(isinf(u) || u!=u);

    for(int itr=0;itr<P_c;itr+=2){

        for(int t=0;t<T;t++){

            for(int i=0;i<N_h;i++){
                //compute for beta,gamma, u
                double x1;
                double x2;
                double min;
                if(population_crossover[itr][t][i]>=population_crossover[itr+1][t][i]){
                    x1 = population_crossover[itr][t][i];
                    x2 = population_crossover[itr+1][t][i];
                }
                if(population_crossover[itr][t][i]<population_crossover[itr+1][t][i]){
                    x1 = population_crossover[itr][t][i];
                    x2 = population_crossover[itr+1][t][i];
                }
                if(population_crossover[itr][t][i]!=population_crossover[itr][t][i] || isinf(population_crossover[itr][t][i])){
                    population_crossover[itr][t][i] = Q_hmax[i];
                    i--;
                    continue;
```

```
}
if(population_crossover[itr+1][t][i]!=population_crossover[itr+1][t][i] || isinf(population_crossover[itr+1][t][i])){
    population_crossover[itr+1][t][i] = Q_hmax[i];
    i--;
    continue;
}
if(x1==x2){
    continue;
}
else{
    if((x1-Q_hmin[i])<(x2-Q_hmax[i])){
        min = x1-Q_hmin[i];
    }
    else{
        min = x2-Q_hmax[i];
    }
    double beta = 1 + ((double)2/(x2-x1))*min;
        if(x2!=x2 || isinf(x2)){
            i--;
            continue;                       }
        if(x1!=x1 || isinf(x1)){
            i--;
            continue;
        }
    if(beta!=beta || isinf(beta)){
        i--;
        continue;
    }
    double asdf = -(n_c+1);
    double alpha;
    if(beta<0){
        alpha = 2 - -pow(fabs(beta),asdf);
    }
    else{
        alpha = 2 - pow(fabs(beta),asdf);
    }
    if(alpha!=alpha || isinf(alpha)){
        i--;
        continue;
    }
    double gamma = 0;
    if(u<=(1/alpha)){
        if((alpha*u)<0){
            gamma = -pow(fabs((alpha*u)),((double)1/(n_c+1)));
        }
        else{
            gamma = pow((alpha*u),((double)1/(n_c+1)));
        }
    }
    else{
        if((1/(2-alpha*u))<0){
            gamma = -pow(fabs(((double)1/(2-alpha*u))),((double)1/(n_c+1)));
        }
        else{
            gamma = pow(((double)1/(2-alpha*u)),((double)1/(n_c+1)));
        }
    }
    if(gamma!=gamma || isinf(gamma)){
        i--;
        continue;
    }

    double y1 = 0.5*((x1+x2) - gamma*fabs((x2-x1)));
```

```
        double y2 = 0.5*((x1+x2) + gamma*fabs((x2-x1)));
        population_crossover[itr][t][i]=y1;
        population_crossover[itr+1][t][i]=y2;
        if(population_crossover[itr][t][i]!=population_crossover[itr][t][i] || isinf(population_crossover[itr][t][i])){
            population_crossover[itr][t][i] = Q_hmax[i];
        }
        if(population_crossover[itr+1][t][i]!=population_crossover[itr+1][t][i] || isinf(population_crossover[itr+1][t][i])){
            population_crossover[itr+1][t][i] = Q_hmax[i];
        }
    }
}

for(int i=N_h;i<N_h+N_s;i++){
    double x1;
    double x2;
    double min;
    if(population_crossover[itr][t][i]>=population_crossover[itr+1][t][i]){
        x1 = population_crossover[itr][t][i];
        x2 = population_crossover[itr+1][t][i];
    }
    if(population_crossover[itr][t][i]<population_crossover[itr+1][t][i]){
        x1 = population_crossover[itr][t][i];
        x2 = population_crossover[itr+1][t][i];
    }
    if(population_crossover[itr][t][i]!=population_crossover[itr][t][i] || isinf(population_crossover[itr][t][i])){
        population_crossover[itr][t][i] = P_smin;
    }
    if(population_crossover[itr+1][t][i]!=population_crossover[itr+1][t][i] || isinf(population_crossover[itr+1][t][i])){
        population_crossover[itr+1][t][i] = P_smin;
    }
    if(x1==x2){
        continue;
    }
    else{
        if((x1-P_smin)<(x2-P_smax)){
            min = x1-P_smin;
        }
        else{
            min = x2-P_smax;
        }
        double beta = 1 + ((double)2/(x2-x1))*min;
        double asdf = -(n_c+1);
        double alpha;
        if(beta<0){
            alpha = 2- -pow(fabs(beta),asdf);
        }
        else{
            alpha = 2- pow(fabs(beta),asdf);
        }
        double gamma = 0;
        if(u<=(1/alpha)){
            if((alpha*u)<0){
                gamma = -pow(fabs((alpha*u)),((double)1/(n_c+1)));
            }
            else{
                gamma = pow((alpha*u),((double)1/(n_c+1)));
            }
        }
        else{
            if((1/(2-alpha*u))<0){
                gamma = -pow(fabs((double)(1/(2-alpha*u))),((double)1/(n_c+1)));
            }
            else{
```

```
                        gamma = pow(((double)1/(2-alpha*u)),((double)1/(n_c+1)));
                    }
                }
                double y1 = 0.5*((x1+x2) - gamma*fabs((x2-x1)));
                double y2 = 0.5*((x1+x2) + gamma*fabs((x2-x1)));
                population_crossover[itr][t][i]=y1;
                population_crossover[itr+1][t][i]=y2;
                if(population_crossover[itr][t][i]!=population_crossover[itr][t][i] || isinf(population_crossover[itr][t][i])){
                    population_crossover[itr][t][i] = P_smin;
                }
                if(population_crossover[itr+1][t][i]!=population_crossover[itr+1][t][i] || isinf(population_crossover[itr+1][t][i])){
                    population_crossover[itr+1][t][i] = P_smin;
                }
            }
        }
    }

    }
}

void crossover(){
    population_crossover.clear();
    deque<int> indexes;//list of indexes for crossover, if picked, pick another
    //pick crossover parents
    for(int i=P_e;i<N_p;i++){//index list for N_p-P_e
        indexes.push_back(i);
    }
    for(int i=0;i<P_c;i++){
        int itr1 = rand()%(N_p-P_e) - i;//itr1 ranges from P_e to N_p
        if(itr1<0) itr1=0;
        population_crossover.push_back(population[indexes[itr1]]);
        indexes.erase(indexes.begin()+(itr1));
    }
    sbx();
}

void mutation(){
    population_mutation.clear();
    deque<int> indexes;//list of indexes for crossover, if picked, pick another
    //pick crossover parents
    for(int i=P_e;i<N_p;i++){//index list for N_p-P_e
        indexes.push_back(i);
    }
    for(int i=0;i<P_m;i++){
        int itr1 = rand()%(N_p-P_e) - i;//itr1 ranges from P_e to N_p
        if(itr1<0) itr1=0;
        population_mutation.push_back(population[indexes[itr1]]);
        indexes.erase(indexes.begin()+(itr1));
    }

    for(int i=0;i<P_m;i++){
        for(int t=0;t<T;t++){
            for(int j=0;j<N_h;j++){
                bool teta = rand()%2;
                if(teta==0){
                    double ra;
                    do{
                        ra = (double)rand()/rand();
                        ra -= (int) ra;
                    }while(isinf(ra) || ra!=ra);

                    double temporary;
                    if(((double)1-((double)t/T))<0){
```

```
                temporary = (double)t/T;
                temporary = 1-temporary;
                temporary = -pow(fabs(temporary), b_mutation);
                temporary = 1-pow(ra,temporary);
                population_mutation[i][t][j] = population_mutation[i][t][j] + (Q_hmax[j]-population_mutation[i][t][j]) * temporary;
            }
            else{
                temporary = (double)t/T;
                temporary = 1-temporary;
                temporary = pow(temporary, b_mutation);
                temporary = 1-pow(ra,temporary);
                population_mutation[i][t][j] = population_mutation[i][t][j] + (Q_hmax[j]-population_mutation[i][t][j]) * temporary;
            }

            if(population_mutation[i][t][j]!=population_mutation[i][t][j] || isinf(population_mutation[i][t][j])){
                population_mutation[i][t][j] = Q_hmax[j];
            }
        }
        if(teta==1){
            double ra;
            do{
                ra = (double)rand()/rand();
                ra -= (int) ra;
            }while(isinf(ra) || ra!=ra);

            double temporary;
            if(((double)1-((double)t/T))<0){
                temporary = (double)t/T;
                temporary = 1-temporary;
                temporary = -pow(fabs(temporary), b_mutation);
                temporary = 1-pow(ra,temporary);
                population_mutation[i][t][j] = population_mutation[i][t][j] + (population_mutation[i][t][j] - Q_hmin[j]) * temporary;
            }
            else{
                temporary = (double)t/T;
                temporary = 1-temporary;
                temporary = pow(temporary, b_mutation);
                temporary = 1-pow(ra,temporary);
                population_mutation[i][t][j] = population_mutation[i][t][j] + (population_mutation[i][t][j] - Q_hmin[j]) * temporary;
            }
            if(population_mutation[i][t][j]!=population_mutation[i][t][j] || isinf(population_mutation[i][t][j])){
                population_mutation[i][t][j] = Q_hmax[j];
            }
        }
    }
    for(int j=N_h;j<N_h+N_s;j++){
        bool teta = rand()%2;
        if(teta==0){
            double ra;
            do{
                ra = (double)rand()/rand();
                ra -= (int) ra;
            }while(isinf(ra) || ra!=ra);

            double temporary;
            if(((double)1-((double)t/T))<0){
                temporary = (double)t/T;
                temporary = 1-temporary;
                temporary = -pow(fabs(temporary), b_mutation);
                temporary = 1-pow(ra,temporary);
                population_mutation[i][t][j] = population_mutation[i][t][j] + (P_smax-population_mutation[i][t][j]) * temporary;
            }
            else{
```

```
                    temporary = (double)t/T;
                    temporary = 1-temporary;
                    temporary = pow(temporary, b_mutation);
                    temporary = 1-pow(ra,temporary);
                    population_mutation[i][t][j] = population_mutation[i][t][j] + (P_smax-population_mutation[i][t][j]) * temporary;
                }
                if(population_mutation[i][t][j]!=population_mutation[i][t][j] || isinf(population_mutation[i][t][j])){
                    population_mutation[i][t][j] = P_smin;
                }
            }
            if(teta==1){
                double ra;
                do{
                    ra = (double)rand()/rand();
                    ra -= (int) ra;
                }while(isinf(ra) || ra!=ra);
                double temporary;
                if(((double)1-((double)t/T))<0){
                    temporary = (double)t/T;
                    temporary = 1-temporary;
                    temporary = -pow(fabs(temporary), b_mutation);
                    temporary = 1-pow(ra,temporary);
                    population_mutation[i][t][j] = population_mutation[i][t][j] + (population_mutation[i][t][j]-P_smin) * temporary;
                }
                else{
                    temporary = (double)t/T;
                    temporary = 1-temporary;
                    temporary = pow(temporary, b_mutation);
                    temporary = 1-pow(ra,temporary);
                    population_mutation[i][t][j] = population_mutation[i][t][j] + (population_mutation[i][t][j]-P_smin) * temporary;
                }
                if(population_mutation[i][t][j]!=population_mutation[i][t][j] || isinf(population_mutation[i][t][j])){
                    population_mutation[i][t][j] = P_smin;
                }
            }
        }
    }
}

void update_population(){
    for(int i=0;i<P_e;i++){//add elite population to P
        population[i].swap(population_elite[i]);
    }
    for(int i=0;i<P_c;i++){//add crossover population to P
        population[P_e+i].swap(population_crossover[i]);
    }
    for(int i=0;i<P_m;i++){
        population[i+P_e+P_c].swap(population_mutation[i]);
    }

}


double euclidian(deque<deque<double> > temp1, deque<deque<double> > temp2){
    double temp=0;
    for(int t=0;t<T;t++){
        for(int j=0;j<N_h+N_s;j++){
            double tempVar = (temp1[t][j] - temp2[t][j]);
                temp = pow(fabs(tempVar),2);
        }
    }
    return sqrt(temp);//return always positive due to square power
```

```
}

double evaluation(deque<deque<double> > temp1){
    double temp=0;
    for(int t=0;t<T;t++){
        for(int i=0;i<N_h+N_s;i++){//uncomment the comment for next line for test system 1, case 2
            temp = temp + a_si + b_si*temp1[t][N_h+i] + c_si*pow(temp1[t][N_h+i],2);// + fabs(d_si*sin(e_si*(P_smin - temp1[t][N_h+i])));
        }
    }
    return temp;
}

double preyS(int index){
    for(int ind = 0;ind<N_p;ind++){
        if(board[index][ind]==1){
            if( evaluation(population[ind])<S_fitness){//random fish is better
                for(int t=0;t<T;t++){
                    for(int j=0;j<N_h+N_s;j++){
                        double ran;
                        do{
                            ran = (double)rand()/rand();
                            ran -=(int) ran;
                        }while(isinf(ran) || ran!=ran);

                        populationS[t][j] = populationS[t][j] + ( ((double)(population[ind][t][j] - populationS[t][j])/euclidian(population[ind],populationS))
                        *step*ran);
                    }
                }
            }
            else{//random fish is not better, perform free move
                for(int t=0;t<T;t++){
                    for(int j=0;j<N_h+N_s;j++){
                        double ran = (double)rand()/rand();
                        ran -= (int) ran;
                        if(isinf(ran) || ran!=ran){
                            j--;
                            continue;
                        }
                        populationS[t][j] = populationS[t][j] + step*ran;
                    }
                }

            }
            return evaluationS();
        }
    }
}

double preyF(int index){
    for(int ind=0;ind<N_p;ind++){
        if(board[index][ind]==1){
            if( evaluation(population[ind])<F_fitness){//random fish is better
                for(int t=0;t<T;t++){
                    for(int j=0;j<N_h+N_s;j++){
                        double ran = (double)rand()/rand();
                        ran -=(int) ran;
                        if(isinf(ran) || ran!=ran){
                            j--;
                            continue;
                        }
                        populationF[t][j] = populationF[t][j] + ( ((double)(population[ind][t][j] - populationF[t][j])/euclidian(population[ind],populationF))
                        *step*ran);
                    }
```

```
                }
            }
            else{//random fish is not better, perform free move
                for(int t=0;t<T;t++){
                    for(int j=0;j<N_h+N_s;j++){
                        double ran = (double)rand()/rand();
                        ran -= (int) ran;
                        if(isinf(ran) || ran!=ran){
                            j--;
                            continue;
                        }
                        populationF[t][j] = populationF[t][j] + step*ran;
                    }
                }
            }
            return evaluationF();
        }
    }
}

double swarm(int index){
    deque<deque<double> > center;
    populationS.clear();
    populationS = population[index];
    S_fitness = population_fitness[index];
    for(int t=0;t<T;t++){//initialize all vectors to zero
        deque<double> a;
        for(int j=0;j<N_h+N_s;j++){
            a.push_back(0);
        }
        center.push_back(a);
        a.clear();
    }
    double N_visual=0;
        for(int itr1=0;itr1<POP_SIZE;itr1++){
            if(board[index][itr1]==1){
                for(int t=0;t<T;t++){
                    for(int j=0;j<N_h+N_s;j++){
                        center[t][j] += population[itr1][t][j];
                    }
                }
                N_visual++;
            }
        }
        if(N_visual==0){
            return preyS(index);
        }
        else{
            for(int t=0;t<T;t++){//divide all center chromosomes to N_visual
                for(int j=0;j<N_h+N_s;j++){
                    center[t][j] = (double)center[t][j]/N_visual;
                }
            }

            //perform swarm behavior
            double euclid = euclidian(populationS,center);
            if(evaluation(center)<(population_fitness[index]*crowd_factor)){//center is better
                for(int t=0;t<T;t++){//apply swarm function
                    for(int j=0;j<N_s+N_h;j++){
                        double ran = (double)rand()/rand();
                        ran -=(int)ran;
                        if(isinf(ran) || ran!=ran){
                            j--;
```

```
                        continue;
                    }
                    populationS[t][j] = populationS[t][j] + ((double)(center[t][j]-populationS[t][j])/euclid)*step*ran;
                }
            }
            return evaluationS();
        }
        else{//current position is better, perform prey behavior
            return preyS(index);
        }
    }
}

double follow(int index){
    populationF = population[index];
    for(int itr1=0;itr1<POP_SIZE;itr1++){//first in deque is already the best, just find the first best visible fish
        if(board[index][itr1]==1){//itr1 fish is visible
            if(itr1>index){//index fish is already the best fish, no follow behavior needed
                return preyF(index);//follow not possible, do prey behavior
            }
            else{//perform follow behavior
                for(int t=0;t<T;t++){
                    for(int j=0;j<N_h+N_s;j++){
                        double ran = (double)rand()/rand();
                        ran -= (int)ran;
                        if(isinf(ran)|| ran!=ran){
                            j--;
                            continue;
                        }
                        populationF[t][j] = populationF[t][j] + (((double)(population[itr1][t][j] - populationF[t][j])/euclidian(populationF,population[itr1]))
                        *step*ran);
                    }
                }
                return evaluationF();
            }
        }
    }
}

void blackboard(){
    //reset
    for(int itr1=0;itr1<POP_SIZE;itr1++){
        for(int itr2=0;itr2<POP_SIZE;itr2++){
            if(itr1==itr2){
                board[itr1][itr2] = 0;
            }
            else{
                board[itr1][itr2]= -1;
            }
        }
    }
    //get visible fishes
    for(int itr1=0;itr1<POP_SIZE;itr1++){
        for(int itr2=0;itr2<POP_SIZE;itr2++){
            if(itr1==itr2){
                continue;
            }
            else{
                if(euclidian(population[itr1],population[itr2])<=visual){
                    board[itr1][itr2] = 1;
                }
            }
        }
```

```
    }
}

void evaluation_pso(){//calculate fitness value for population
    population_pso_fitness.clear();
    for(int itr=0;itr<POP_SIZE;itr++){
        double temp=0;
        for(int t=0;t<T;t++){
            for(int i=0;i<N_s;i++){//uncomment the comment on next line for test system 1, case 2
                temp = temp + a_si + b_si*population_pso[itr][t][N_h+i] + c_si*pow(population_pso[itr][t][N_h+i],2);
                // + fabs(d_si*sin(e_si*(P_smin - population[itr][t][N_h+i])));
            }
        }

        population_pso_fitness.push_back(temp);//default place for the fitness value
    }
}

void initialize_pso(){//initialize all velocity to 0
    for(int i=0;i<N_p;i++){
        deque<deque<double> > temp1;
        for(int t=0;t<T;t++){
            deque<double> temp2;

            for(int j=0;j<N_h;j++){
                bool ran = rand()%2;
                double temp3 = abs(static_cast<double> (rand() / static_cast<double> (RAND_MAX/(Q_hmin[j] - Q_hmax[j]) )));
                if(ran==true){
                    temp3 = -temp3;
                }
                if(isinf(temp3) || temp3!=temp3){
                    j--;
                    continue;
                }
                temp2.push_back(temp3);
            }

            for(int j=N_h;j<N_h+N_s;j++){
                bool ran = rand()%2;
                double temp3 = abs(static_cast<double> (rand() / static_cast<double> (RAND_MAX/(Q_hmin[j] - Q_hmax[j]) )));
                if(ran==true){
                    temp3 = -temp3;
                }
                if(isinf(temp3) || temp3!=temp3){
                    j--;
                    continue;
                }
                temp2.push_back(temp3);
            }

            temp1.push_back(temp2);
            temp2.clear();
        }
            velocity.push_back(temp1);
            temp1.clear();
    }

    for(int i=0;i<N_p;i++){//copy population to pbest
        deque<deque<double> > temp1;
        for(int t=0;t<T;t++){
            deque<double> temp2;
            for(int j=0;j<N_h+N_s;j++){
                temp2.push_back(population[i][t][j]);
```

```
            }
            temp1.push_back(temp2);
            temp2.clear();
        }
        pbest.push_back(temp1);
        temp1.clear();
        pbest_fitness.push_back(population_fitness[i]);
    }
        //initialize population_pso to zeros
    for(int i=0;i<N_p;i++){
        deque<deque<double> > temp1;
        for(int t=0;t<T;t++){
            deque<double> temp2;
            for(int j=0;j<N_h+N_s;j++){
                temp2.push_back(0);
            }
            temp1.push_back(temp2);
            temp2.clear();
        }
        population_pso.push_back(temp1);
        temp1.clear();
    }
        //get first as gbest
        gbest = population[0];
        gbest_fitness = population_fitness[0];
}

double update_velocity(int i, int t, int j, int itr){//returns updated velocity of i t j
        double rp, rg;
        double varphi_p = 2.05;
        double varphi_g = 2.1;
        double C;
        double w;
        double wmin = 0.1;
        double wmax = 1.0;
        double varphi = varphi_g+varphi_g;
        C = (double)2/(2-varphi - sqrt(pow(varphi,2) - 4*varphi ));
        w = (double)(wmax-wmin)*((double)(gmax_pso - itr)/gmax_pso) + wmin;
        do{
            rp = (double)rand()/rand();
            rp -= (int)rp;
        }while(isinf(rp) || rp!=rp);
        do{
            rg = (double)rand()/rand();
            rg -= (int)rg;
        }while(isinf(rg) || rg!=rg);
        double newv = (C*(w*velocity[i][t][j] + rp*varphi_p*(pbest[i][t][j] - population[i][t][j]) + rg*varphi_g*(gbest[t][j] - population[i][t][j]) ) );
        return newv;
}

void pso(int itr){//run pso for all population
    for(int i=0;i<N_p;i++){
        for(int t=0;t<T;t++){
            for(int j=0;j<N_h+N_s;j++){
                velocity[i][t][j] = update_velocity(i,t,j,itr);
                population_pso[i][t][j] = population[i][t][j] + velocity[i][t][j];
            }
        }
    }
    //evaluation
    evaluation_pso();

    //update pbest
```

```
    for(int i=0;i<N_p;i++){
        if(population_fitness[i]<pbest_fitness[i]){
            for(int t=0;t<T;t++){
                for(int j=0;j<N_h+N_s;j++){
                    pbest[i][t][j] = population[i][t][j];
                }
            }
            pbest_fitness[i] = population_fitness[i];
        }
        else{
            continue;
        }
    }

    //update population if necessary from population_pso
    for(int i=0;i<N_p;i++){
        if(population_pso_fitness[i] < population_fitness[i]){
            population[i].swap(population_pso[i]);
            population_fitness[i] = population_pso_fitness[i];
        }
    }
    double bestIndex;
    for(int i=0;i<N_p;i++){
        double temp = 999999999999999;
        if(population_fitness[i]<temp){
            temp = population_fitness[i];
            bestIndex = i;
        }
    }
    gbest_fitness = population_fitness[bestIndex];
    for(int t=0;t<T;t++){
        for(int j=0;j<N_h+N_s;j++){
            gbest[t][j] = population[bestIndex][t][j];
        }
    }
}

double update_velocity1(int i, int t, int j, int itr){//returns updated velocity of i t j
        double c1, c2;
        double c1i = 2.5;
        double c1f = 0.5;
        double c2i = 0.5;
        double c2f = 2.5;
        double rand1, rand2, rand3;
        double newv;
        do{
            rand1 = (double)rand()/rand();
            rand1 -= (int)rand1;
        }while(isinf(rand1) || rand1!=rand1);
        do{
            rand2 = (double)rand()/rand();
            rand2 -= (int)rand2;
        }while(isinf(rand2) || rand2!=rand2);
        do{
            rand3 = (double)rand()/rand();
            rand3 -= (int)rand3;
        }while(isinf(rand3) || rand3!=rand3);

        if(velocity[i][t][j]==0 && rand3<0.5){
            double rand4;
            do{
            rand4 = (double)rand()/rand();
            rand4 -= (int)rand4;
```

```
            }while(isinf(rand4) || rand4!=rand4);
            if(j<4){//vmax is for hydro
                newv = rand4*Q_hmax[j];
            }
            else{//vmax is for thermal
                newv = rand4*P_smax;
            }
            return newv;
        }
        else{
            double rand5;
            do{
            rand5 = (double)rand()/rand();
            rand5 -= (int)rand5;
            }while(isinf(rand5) || rand5!=rand5);
            if(j<4){//vmax is for hydro
                newv = -rand5*Q_hmax[j];
            }
            else{//vmax is for thermal
                newv = -rand5*P_smax;
            }
            return newv;
        }
        c1 = (c1f-c1i)*(double)(itr/gmax_pso) + c1i;
        c2 = (c2f-c2i)*(double)(itr/gmax_pso) + c2i;
        newv = c1*rand1*(pbest[i][t][j] - population[i][t][j]) + c2*rand2*(gbest[t][j] - population[i][t][j]);
        return newv;
}

void sohpso(int itr){//run pso for all population
    for(int i=0;i<N_p;i++){
        for(int t=0;t<T;t++){
            for(int j=0;j<N_h+N_s;j++){
                velocity[i][t][j] = update_velocity1(i,t,j,itr);
                population_pso[i][t][j] = population[i][t][j] + velocity[i][t][j];
            }
        }
    }
    //evaluation
    evaluation_pso();

    //update pbest
    for(int i=0;i<N_p;i++){
        if(population_fitness[i]<pbest_fitness[i]){
            for(int t=0;t<T;t++){
                for(int j=0;j<N_h+N_s;j++){
                    pbest[i][t][j] = population[i][t][j];
                }
            }
            pbest_fitness[i] = population_fitness[i];
        }
        else{
            continue;
        }
    }

    for(int i=0;i<N_p;i++){
        if(population_pso_fitness[i] < population_fitness[i]){
            population[i].swap(population_pso[i]);
            population_fitness[i] = population_pso_fitness[i];
        }
    }
    //update gbest
```

```
    double bestIndex;
    for(int i=0;i<N_p;i++){
        double temp = 999999999999999;
        if(population_fitness[i]<temp){
            temp = population_fitness[i];
            bestIndex = i;
        }
    }
    gbest_fitness = population_fitness[bestIndex];
    for(int t=0;t<T;t++){
        for(int j=0;j<N_h+N_s;j++){
            gbest[t][j] = population[bestIndex][t][j];
        }
    }
}


void constraints_handling(){
    for(int i=0;i<POP_SIZE;i++){
        for(int j=0;j<N_h;j++){//constraints handling for hydro plants
            int iteration;
            iteration = 0;
            check_discharge(i,j);//check Q_h(j,t)
            double V_hc = volume[i][T-1][j] - V_hend[j];
            while(fabs(V_hc)>e_vcourse && iteration<=iteration_course){
                double V_haverage = V_hc/T;
                for(int t=0;t<T;t++){
                    population[i][t][j] = population[i][t][j] + V_haverage;
                    if(population[i][t][j]!=population[i][t][j] || isinf(population[i][t][j])){
                        population[i][t][j] = Q_hmax[j];
                    }
                    check_discharge1(i,j,t);
                }
                update_volume(i);
                V_hc = volume[i][T-1][j] - V_hend[j];
                iteration++;
            }
            iteration=0;
            deque<double> Q_hc;
            while(fabs(V_hc)>e_vfine && iteration<iteration_fine){
                Q_hc.clear();
                Q_hc = Q_hcF(i,j);//get the change in water discharge population from i and j
                int itr = max_adjustableQ(Q_hc);
                population[i][itr][j] = population[i][itr][j] + V_hc;
                if(population[i][itr][j]!=population[i][itr][j] || isinf(population[i][itr][j])){
                    population[i][itr][j] = Q_hmax[j];
                }
                update_volume(i);
                check_discharge1(i,j,itr);
                V_hc = volume[i][T-1][j] - V_hend[j];
                iteration++;
            }
        }
        update_volume(i);
        for(int t=0;t<T;t++){//thermal plants constraints handling
            int iteration = 0;
            double P_sc = P_D[t] - P_ssum(i,t) - P_hsum(i,t);//no transmission loss
            population[i][t][4] += P_sc;
        }
    }
    for(int i=0;i<POP_SIZE;i++){
        for(int t=0;t<T;t++){
            for(int j=0;j<N_h;j++){
```

```
                if(isinf(population[i][t][j]) || (population[i][t][j]!=population[i][t][j])){
                    population[i][t][j] = Q_hmax[j];
                }
            }
            for(int j=N_h;j<N_h+N_s;j++){
                if(isinf(population[i][t][j]) || (population[i][t][j]!=population[i][t][j])){
                    population[i][t][j] = P_smin;
                }
            }
        }
    }
}


int main(){
best_fitness = 999999999999999;
inflowF();
initUpstream();
aveTime = 0;
for(int t=0;t<T;t++){//initialization of super global variable
    deque<double> temps;//chromosome
    deque<double> temps1;//Ph Qh
    deque<double> temps2;//Ps
    if(t==0){
        for(int j=0;j<N_h+N_s;j++){
            temps.push_back(0);
        }
        for(int j=0;j<N_h;j++){
            temps1.push_back(0);
        }
        for(int j=N_h;j<N_h+N_s;j++){
            temps2.push_back(0);
        }
    }
    aveBestChromosome.push_back(temps);
    aveWorstChromosome.push_back(temps);
    aveBestPh.push_back(temps1);
    aveBestPs.push_back(temps2);
    aveBestQh.push_back(temps1);
}



for(int mainItr=0;mainItr<iterators;mainItr++){
    population.clear();
    population_fitness.clear();
    volume.clear();
    population_elite.clear();
    population_crossover.clear();
    population_mutation.clear();
    populationS.clear();
    populationF.clear();
    S_fitness = 0;
    F_fitness = 0;
    best_fitness = 9999999999999999;
    srand(time(NULL));
    //INITIALIZATION
    for(int i=0;i<POP_SIZE;i++){//temp(n), n - dimension
        deque<deque<double> > temp2;
        for(int t=0;t<T;t++){
            deque<double> temp1;

            for(int nh=0;nh<N_h;nh++){
```

```
            double ran;
            double temp;
            do{
                ran = (double)rand()/rand();
                ran -= (int) ran;
            }while(isinf(ran) || ran!=ran);
            temp = Q_hmin[nh] + ran*(Q_hmax[nh]-Q_hmin[nh]);
            temp1.push_back(temp);
        }

        for(int ns=N_h;ns<N_h+N_s;ns++){
            double ran;
            double temp;
            do{
                ran = (double)rand()/rand();
                ran -= (int) ran;
            }while(isinf(ran) || ran!=ran);
            temp = P_smin + ran*(P_smax-P_smin);
            temp1.push_back(temp);
        } temp2.push_back(temp1);
        temp1.clear();
    }
    population.push_back(temp2);
    temp2.clear();
}


for(int i=0;i<POP_SIZE;i++){//water dynamic volume generation
    //water dynamic balance generation
    deque<deque<double> > V_h;//V_h(t,j)
    V_h.clear();
    for(int t=0;t<T;t++){
        deque<double> temp1;
        for(int j=0;j<N_h;j++){
            double temp=0;//V_h(j)
            if((t-1)<0){
                temp = V_hbegin[j];// + inflow[j][t] - population[i][t][j] + sum_upstream(i,t,j);//spillage not calculated
                temp1.push_back(temp);
            }
            else{
                temp = V_h[t-1][j] + inflow[j][t] - population[i][t][j] + sum_upstream(i,t,j);//spillage not calculated
                temp1.push_back(temp);
            }
        }
        V_h.push_back(temp1);
        temp1.clear();
    }
    volume.push_back(V_h);//always update volume reservoir after change in values
    V_h.clear();
}


//CONSTRAINTS HANDLING
for(int i=0;i<POP_SIZE;i++){
    for(int j=0;j<N_h;j++){//constraints handling for hydro plants
        int iteration;
        iteration = 0;
        check_discharge(i,j);//check Q_h(j,t)
        double V_hc = volume[i][T-1][j] - V_hend[j];
        while(fabs(V_hc)>e_vcourse && iteration<iteration_course){
            double V_haverage = V_hc/T;
            for(int t=0;t<T;t++){
                population[i][t][j] = population[i][t][j] + V_haverage;
```

```
                check_discharge1(i,j,t);
            }
            update_volume(i);
            V_hc = volume[i][T-1][j] - V_hend[j];
            iteration++;
        }
        iteration=0;
        deque<double> Q_hc;
        while(fabs(V_hc)>e_vfine && iteration<iteration_fine){
            Q_hc.clear();
            Q_hc = Q_hcF(i,j);//get the population from i and j, change in Q
            int itr = max_adjustableQ(Q_hc);
            population[i][itr][j] = population[i][itr][j] + V_hc;
            update_volume(i);
            check_discharge1(i,j,itr);
            V_hc = volume[i][T-1][j] - V_hend[j];
            iteration++;
        }
    }
    update_volume(i);

    for(int t=0;t<T;t++){//thermal plants constraints handling
        int iteration = 0;
        double P_sc = P_D[t] - P_ssum(i,t) - P_hsum(i,t);//no transmission loss
        population[i][t][4] +=P_sc;
    }
}
clock_t t;
t = clock();
//RCGA
for(int g = 0;g<gmax_rcga;g++){
    //EVALUATION
    evaluation();//generate fitness value and sort them

    elitist();//copy all the elite population, get best_fitness

    //CROSSOVER
    crossover();//pure crossover w/ sbx.

    //MUTATION
    mutation();//pure mutation

    update_population();//add the crossover, mutated population to the general population

    constraints_handling();

    population_fitness.clear();
    population_elite.clear();
    population_crossover.clear();
    population_mutation.clear();
}
constraints_handling();


//AFSA
for(int g = 0;g<gmax_afsa;g++){
    evaluation();
    for(int i=0;i<POP_SIZE;i++){//perform evaluation and ranking after every functions and method
        best_fitness = population_fitness[0];
        double swarmVar = swarm(i);//swarmVar is the fitness value of ith fish that performed swarm
        double followVar = follow(i);//followVar is the fitness value of ith fish that performed follow
        if(swarmVar>followVar){//choose follow behavior
            population[i].swap(populationF);
```

```
            population_fitness[i] = F_fitness;
        }
        else{//choose swarm behavior
            population[i].swap(populationS);
            population_fitness[i] = S_fitness;
        }
    }
    constraints_handling();
    blackboard();
}

//PSO
initialize_pso();
for(int g = 0;g<gmax_pso;g++){
    pso(g);
    //constraints handling for pso
    constraints_handling();
}
//SOHPSO
    initialize_pso();
for(int g = 0;g<gmax_pso;g++){
    sohpso(g);
    //constraints handling for sohpso
    constraints_handling();
}

t=clock() - t;

evaluation();
cout<<"----------------------------------------------------------------"<<endl;
aveBestFitness.push_back(population_fitness[0]);
aveWorstFitness.push_back(population_fitness[99]);

cout<<"Best Fitness: "<<population_fitness[0]<<endl;
cout<<"Worst Fitness: "<<population_fitness[99]<<endl;

for(int t=0;t<T;t++){
    for(int j=0;j<N_h+N_s;j++){
        aveBestChromosome[t][j]+=population[0][t][j];
    }
}
for(int t=0;t<T;t++){
    for(int j=0;j<N_h+N_s;j++){
        aveWorstChromosome[t][j]+=population[99][t][j];
    }
}
cout<<"----------------------------------------------------------------"<<endl;
cout<<"P_h: "<<endl;
for(int t=0;t<T;t++){//add average best P_h
    for(int a=0;a<N_h;a++){
        double temp = 0;
        temp = c1[a]*pow(volume[0][t][a],2) + c2[a]*pow(population[0][t][a],2) + c3[a]*volume[0][t][a]*population[0][t][a] + c4[a]*volume[0][t][a] + c5[a]*popula
        if(temp<0){
            temp=0;
        }cout<<temp<<" ";
        aveBestPh[t][a] += temp;
    }
    cout<<endl;
}
cout<<"----------------------------------------------------------------"<<endl;
for(int t=0;t<T;t++){
    for(int j=N_h;j<N_h+N_s;j++){
        aveBestPs[t][j] += population[0][t][j];
```

```
        }
    }

    for(int t=0;t<T;t++){
        for(int j=0;j<N_h;j++){
            aveBestQh[t][j] += population[0][t][j];
        }
    }

    cout<<"Best: "<<endl;
    for(int i=0;i<1;i++){
        for(int t=0;t<T;t++){
            for(int j=0;j<N_h+N_s;j++){
                cout<<population[i][t][j]<<" ";
            }
            cout<<endl;
        }
        cout<<endl;
    }
    cout<<endl;


    cout<<"Volume: "<<endl;
    for(int i=0;i<2;i++){
        for(int t=0;t<T;t++){
            for(int j=0;j<N_h;j++){
                cout<<volume[i][t][j]<<" ";
            }
            cout<<endl;
        }
        cout<<endl;
    }
    cout<<endl;
    cout<<"----------------------------------------------------------------------"<<endl;
    aveTime += (float)t/CLOCKS_PER_SEC;
    cout<<(float)t/CLOCKS_PER_SEC<<" secs"<<endl;
}//end of main iteration (30 runs)


for(int t=0;t<T;t++){
    for(int j=0;j<N_h+N_s;j++){
        aveBestChromosome[t][j] = (double)aveBestChromosome[t][j]/iterators;
        aveWorstChromosome[t][j] = (double)aveWorstChromosome[t][j]/iterators;
    }
}

for(int t=0;t<T;t++){
    for(int j=0;j<N_h;j++){
        aveBestPh[t][j] = (double)aveBestPh[t][j]/iterators;
        aveBestQh[t][j] = (double)aveBestQh[t][j]/iterators;
    }
}

for(int t=0;t<T;t++){
    for(int j=0;j<N_s;j++){
        aveBestPs[t][j] = (double)aveBestPs[t][j]/iterators;
    }
}

double aveBestF=0;
double aveWorstF=0;
for(int i=0;i<30;i++){
    aveBestF += aveBestFitness[i];
```

```
    aveWorstF += aveWorstFitness[i];
}
aveBestF = (double)aveBestF/iterators;
aveWorstF = (double)aveWorstF/iterators;

cout<<"Average Best: "<<aveBestF<<endl;
cout<<"Average Worst: "<<aveWorstF<<endl;
cout<<endl;
cout<<"============================================================"<<endl;
cout<<"Average Best Chromosome: "<<endl;
for(int t=0;t<T;t++){
    for(int j=0;j<N_h+N_s;j++){
        cout<<aveBestChromosome[t][j]<<" ";
    }
    cout<<endl;
}
cout<<"============================================================"<<endl;
cout<<"Average Worst Chromosome: "<<endl;
for(int t=0;t<T;t++){
    for(int j=0;j<N_h+N_s;j++){
        cout<<aveWorstChromosome[t][j]<<" ";
    }
    cout<<endl;
}
cout<<"============================================================"<<endl;
cout<<"Average Best P_h: "<<endl;
for(int t=0;t<T;t++){
    for(int j=0;j<N_h;j++){
        cout<<aveBestPh[t][j]<<" ";
    }
    cout<<endl;
}
cout<<"============================================================"<<endl;
cout<<"Average Best P_s"<<endl;
for(int t=0;t<T;t++){
    for(int j=0;j<N_s;j++){
        cout<<aveBestPs[t][j]<<" ";
    }
    cout<<endl;
}
cout<<"============================================================"<<endl;
cout<<"Total Generation:"<<endl;
for(int t=0;t<T;t++){
    double temp = 0;
    for(int j=0;j<N_h;j++){
        temp += aveBestPh[t][j];
    }
    for(int j=N_h;j<N_h+N_s;j++){
        temp += aveBestChromosome[t][j];
    }
    cout<<temp<<endl;
}
cout<<"============================================================"<<endl;
cout<<"Average Best Q_h"<<endl;
for(int t=0;t<T;t++){
    for(int j=0;j<N_h;j++){
        cout<<aveBestQh[t][j]<<" ";
    }
    cout<<endl;
}
cout<<"============================================================"<<endl;
```

```
return 0;
}
```

Appendix H

Source Code for Test System 2, all cases

```
#include <iostream>
#include <deque>
#include <cstdlib>
#include <ctime>
#include <cmath>
#define POP_SIZE 100
//TEST SYSTEM 2
//Constants
double T=24; //time interval
double iterators = 30; //run iterations
int N_h = 4; //number of hydro plants
int N_s = 3; //number of thermal plants
int N_p = 100; //total number of population
int P_e = 30; //total number of elite individuals in N_p
int P_c = 30; //population from crossover
int P_m = 40; //population number of mutated individuals
double step = .5; //step for AFSA
double visual = 1; //visual for AFSA
double e_vcourse = 1; //for constraints handling
double e_vfine = .01;//for constraints handling
double e_pcourse = 2;//
double e_pfine = .01;
double iteration_course = 20;
double iteration_fine = 10;
double P_smin[3] = {20,40,50};
double P_smax[3] = {176,300,500};
double Q_hmin[4] = {5,6,10,13};
double Q_hmax[4] = {15,15,30,25};
double V_hmin[4] = {80,60,100,170};
double V_hmax[4] = {150,120,240,160};
double V_hbegin[4] = {100,80,170,120};
double V_hend[4] = {120,70,170,140};
double a_si[3] = {100,120,150};
double b_si[3] = {2.45,2.32,2.10};
double c_si[3] = {0.0012,0.0010,0.0015};
double d_si[3] = {160,180,200};
double e_si[3] = {0.038, 0.037,0.035};
double UR[2] = {0,0};
double DR[2] = {0,0};
double R_u[4] = {0,0,2,1};
double P_D[24] = {750, 780, 700, 650, 670, 800, 950, 1010, 1090, 1080, 1100, 1150, 1110, 1030, 1010, 1060, 1050, 1120, 1070, 1050, 910, 860, 850, 800};
double time_delay[4] = {2,3,4,0};
//change the iteration number depending on the hybrid. 0 to disable method
double gmax_rcga = 1000;
double gmax_afsa = 400;
double gmax_pso = 400;
double gmax_sohpso = 400;
double b_mutation = 1;
double n_c = 1;
double c1[4] = {-0.0042, -0.0040, -0.0016, -0.0030};
double c2[4] = {-0.42, -0.30, -0.30, -0.31};
double c3[4] = {0.030, .015, 0.014, 0.027};
```

```
double c4[4] = {0.90, 1.14, 0.55, 1.44};
double c5[4] = {10, 9.5, 5.5, 14};
double c6[4] = {-50, -70, -40, -90};
double inflow[4][24];
double B[7][7];
double B0[7] = {-0.00000075, -0.00000006, 0.00000070, -0.00000003, 0.00000027, -0.00000077, -0.00000001};
double B00 = .55;
double crowd_factor = .8;
int upstreamIndex[4][4];
using namespace std;
deque<deque<deque<double> > > population;// the last element in 3d deque is the fitness value
deque<double> population_fitness;
deque<deque<deque<double> > > volume;
deque<deque<deque<double> > > population_elite;
deque<deque<deque<double> > > population_crossover;
deque<deque<deque<double> > > population_mutation;
int board[POP_SIZE][POP_SIZE];//map, list of visible fishes
deque<deque<double> > populationS;
deque<deque<double > >  populationF;
double S_fitness=0;
double F_fitness=0;
double bulletin;
double best_fitness;


//SUPER UNIVERSAL VARIABLES
deque<double> aveBestFitness;
deque<double> aveWorstFitness;
deque<deque<double> > aveBestChromosome;
deque<deque<double> > aveWorstChromosome;
deque<deque<double> > aveBestPh;
deque<deque<double> > aveBestPs;
deque<deque<double> > aveBestQh;
double aveTime;

void inflowF(){
    double huehue1[24] = {10, 9, 8, 7, 6, 7, 8, 9, 10, 11, 12, 10, 11, 12, 11, 10, 9, 8, 7, 6, 7, 8, 9, 10};
    double huehue2[24] = {8, 8, 9, 9, 8, 7, 6, 7, 8, 9, 9, 8, 8, 9, 9, 8, 7, 6, 7, 8, 9, 9, 8, 8};
    double huehue3[24] = {8.1, 8.2, 4, 2, 3, 4, 3, 2, 1, 1, 1, 2, 4, 3, 3, 2, 2, 2, 1, 1, 2, 2, 1, 0};
    double huehue4[24] = {2.8, 2.4, 1.6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    for(int i=0;i<24;i++){
        inflow[0][i] = huehue1[i];
    }
    for(int i=0;i<24;i++){
        inflow[1][i] = huehue2[i];
    }
    for(int i=0;i<24;i++){
        inflow[2][i] = huehue3[i];
    }
    for(int i=0;i<24;i++){
        inflow[3][i] = huehue4[i];
    }
    double hyu1[7] = {0.0000034, 0.0000013, 0.0000009, -0.0000001, -0.0000008, -0.0000001, -0.0000002};
    double hyu2[7] = {0.0000013, 0.0000014, 0.0000010, 0.0000001, -0.0000005, -0.0000002, -0.0000001};
    double hyu3[7] = {0.0000009, 0.0000010, 0.0000031, 0.0000000, -0.0000011, -0.0000007, -0.0000005};
    double hyu4[7] = {-0.0000001, 0.0000001, 0.0000000, 0.0000024, -0.0000008, -0.0000004, -0.0000007};
    double hyu5[7] = {-0.0000008, -0.0000005, -0.0000011, -0.0000008, 0.0000192, 0.0000027, -0.0000002};
    double hyu6[7] = {-0.0000001, -0.0000002, -0.0000007, -0.0000004, 0.0000027, 0.0000032, 0.0000000};
    double hyu7[7] = {-0.0000002, -0.0000001, -0.0000005, -0.0000007, -0.0000002, 0.0000000, 0.0000135};
    for(int i=0;i<7;i++){
        B[0][i] = hyu1[i];
    }
    for(int i=0;i<7;i++){
```

```
        B[1][i] = hyu2[i];
    }
    for(int i=0;i<7;i++){
        B[2][i] = hyu3[i];
    }
    for(int i=0;i<7;i++){
        B[3][i] = hyu4[i];
    }
    for(int i=0;i<7;i++){
        B[4][i] = hyu5[i];
    }
    for(int i=0;i<7;i++){
        B[5][i] = hyu6[i];
    }
    for(int i=0;i<7;i++){
        B[6][i] = hyu7[i];
    }
}

void initUpstream(){//initialize what plant is upstream of the jth hydro plant, if -1, no plant upstream
    for(int i=0;i<N_h;i++){
        for(int j=0;j<N_h;j++){
            upstreamIndex[i][j] = -1;
        }
    }
    upstreamIndex[2][0] = 0;
    upstreamIndex[2][1] = 1;
    upstreamIndex[3][0] = 2;
}

double sum_upstream(int pop,int t,int j){
    double temp=0;
    for(int i=0;i<R_u[j];i++){
        if((t-time_delay[i])<0){
            continue;
        }
        else{
            if(isinf(population[pop][t-time_delay[i]][j]) || population[pop][t-time_delay[i]][j]!=population[pop][t-time_delay[i]][j]){
                population[pop][t-time_delay[i]][j] = Q_hmax[j];
                continue;
            }
            else{//////////////////////////////////////////
                    temp = temp + population[pop][t-time_delay[i]][upstreamIndex[j][i]];
            }
        }
    }
    return temp;
}

void check_discharge(int index,int j){//check every Q_h for all t in jth hydro plant
    for(int t=0;t<T;t++){
        if(population[index][t][j]<Q_hmin[j]){
            population[index][t][j] = Q_hmin[j];
            if(population[index][t][j]!=population[index][t][j] || isinf(population[index][t][j])){
                population[index][t][j] = Q_hmax[j];
            }
        }
        if(population[index][t][j]>Q_hmax[j]){
            population[index][t][j] = Q_hmax[j];
            if(population[index][t][j]!=population[index][t][j] || isinf(population[index][t][j])){
                population[index][t][j] = Q_hmax[j];
            }
        }
```

```
    }
}

void check_discharge1(int index,int j, int t){//check every Q_h for all t in jth hydro plant
        if(population[index][t][j]<Q_hmin[j]){
            population[index][t][j] = Q_hmin[j];
            if(population[index][t][j]!=population[index][t][j] || isinf(population[index][t][j])){
                population[index][t][j] = Q_hmax[j];
            }
        }
        if(population[index][t][j]>Q_hmax[j]){
            population[index][t][j] = Q_hmax[j];
            if(population[index][t][j]!=population[index][t][j] || isinf(population[index][t][j])){
                population[index][t][j] = Q_hmax[j];
            }
        }
}

void check_thermal1(int index, int t, int j){
    if(population[index][t][j] <=P_smin[j-N_h]){
        population[index][t][j] = P_smin[j-N_h];
        if(population[index][t][j]!=population[index][t][j]){
            }
    }
    if(population[index][t][j] >= P_smax[j-N_h]){
        population[index][t][j] = P_smax[j-N_h];
        if(population[index][t][j]!=population[index][t][j]){
            }
    }
}

deque<double> Q_hcF(int index,int j){
    deque<double> temp;
    for(int t=0;t<T;t++){
        if(t==0){
            temp.push_back(population[index][t][j] - Q_hmin[j]);
        }
        else{
            temp.push_back((population[index][t][j]-population[index][t-1][j]));
            if(population[index][t][j]!=population[index][t][j] || population[index][t-1][j]!=population[index][t-1][j] ||
            isinf(population[index][t][j]) || isinf(population[index][t-1][j])){
                population[index][t][j] = Q_hmax[j];
                population[index][t-1][j] = Q_hmax[j];
            }

        }

    }
    return temp;
}

int max_adjustableQ(deque<double> input){//do just like with max_adjustableP(), but with respect to bounds
    int itr=0;
    double temp=99999999999999999;
    for(int i=1;i<T;i++){
        if(temp<input[i]){
            temp = input[i];
            itr=i;
        }
    }
    return itr;
}
```

```
int max_adjustableP(int index, int t){
    double temp = 99999999999999999;
    int adIndex=N_h;
    deque<double> adjustable;
    for(int itr1=N_h;itr1<N_h+N_s;itr1++){
        if(t==0){//if initial reading, minimum are used as previous
            adjustable.push_back(population[index][t][itr1]-P_smin[itr1-N_h]);
        }
        else{
            adjustable.push_back(population[index][t][itr1]-population[index][t-1][itr1]);
        }

    }
    for(int i=0;i<N_s;i++){
        if(temp>adjustable[i]){//i is the maximum adjustable
            temp=adjustable[i];
            adIndex = i+N_h;
        }
    }
    return adIndex;
}

double P_ssum(int index, int t){
    double temp=0;
    for(int a = N_h; a<N_h+N_s;a++){
        if(population[index][t][a]<0){
            continue;
        }
        temp = temp + population[index][t][a];

    }
    if(temp<0){
        temp =0;
    }
    return temp;
}

double P_hsum(int index, int t){//work on the equation
    double temp=0;
    for(int a = 0; a<N_h;a++){
        double temp1 = c1[a]*pow(volume[index][t][a],2) + c2[a]*pow(population[index][t][a],2) + c3[a]*volume[index][t][a]*population[index][t][a]
         + c4[a]*volume[index][t][a] + c5[a]*population[index][t][a] + c6[a];
        if(temp1<0){
            continue;
        }
        temp = temp + temp1;
    }
    if(temp<0){
        temp = 0;
    }
    return temp;
}

void update_volume(){
    for(int i=0;i<POP_SIZE;i++){
        deque<deque<double> > V_h;//V_h(t,j)
        V_h.clear();
        for(int t=0;t<T;t++){
            deque<double> temp1;
            for(int j=0;j<N_h;j++){
                double temp=0;//V_h(j)
                if((t-1)<0){
                    temp = V_hbegin[j];// + inflow[j][t] - population[i][t][j] + sum_upstream(i,t,j);//spillage not calculated
```

```
                    temp1.push_back(temp);
                }
                else{
                    temp = V_h[t-1][j] + inflow[j][t] - population[i][t][j] + sum_upstream(i,t,j);//spillage not calculated
                    temp1.push_back(temp);
                }
            }
            V_h.push_back(temp1);
            temp1.clear();
        }
        volume[i].swap(V_h);
        V_h.clear();
    }

}

void update_volume(int index){
    deque<deque<double> > V_h;//V_h(t,j)
        V_h.clear();
        for(int t=0;t<T;t++){
            deque<double> temp1;
            for(int j=0;j<N_h;j++){
                double temp;//V_h(j)
                if((t-1)<0){
                    temp = V_hbegin[j];// + inflow[j][t] - population[index][t][j] + sum_upstream(index,t,j);//spillage not calculated
                    temp1.push_back(temp);
                }
                else{
                    temp = V_h[t-1][j] + inflow[j][t] - population[index][t][j] + sum_upstream(index,t,j);//spillage not calculated
                    temp1.push_back(temp);

                }
            }
            V_h.push_back(temp1);
            temp1.clear();
        }
        volume[index].swap(V_h);
        V_h.clear();
}

void evaluation(){//calculate fitness value for population
    population_fitness.clear();
    for(int itr=0;itr<POP_SIZE;itr++){
        double temp=0;
        for(int t=0;t<T;t++){
            double temp2 = 0;
            for(int i=0;i<N_s;i++){
                double temp1 = a_si[i] + b_si[i]*population[itr][t][N_h+i] + c_si[i]*pow(population[itr][t][N_h+i],2)
                + fabs(d_si[i]*sin(e_si[i]*(P_smin[i] - population[itr][t][N_h+i])));
                temp+=temp1;
            }

        }
        population_fitness.push_back(temp);//default place for the fitness value
    }

    //bubble sort
    for(int itr1=0;itr1<POP_SIZE;itr1++){
        for(int itr2=0;itr2<POP_SIZE;itr2++){
            if(population_fitness[itr1]<population_fitness[itr2]){
                population[itr1].swap(population[itr2]);
                double huehue = population_fitness[itr1];
                population_fitness[itr1] = population_fitness[itr2];
```

```
                population_fitness[itr2] = huehue;
            }
        }
    }
}

double evaluationS(){//calculate fitness value for populationF
        double temp=0;
        for(int t=0;t<T;t++){
            for(int i=0;i<N_s;i++){
                temp = temp + a_si[i] + b_si[i]*populationS[t][N_h+i] + c_si[i]*pow(populationS[t][N_h+i],2)
                 + fabs(d_si[i]*sin(e_si[i]*(P_smin[i] - populationS[t][N_h+i])));
            }

        }
        S_fitness = temp;//default place for the fitness value
        return S_fitness;

}

double evaluationF(){//calculate fitness value for populationF
        double temp=0;
        for(int t=0;t<T;t++){
            for(int i=0;i<N_s;i++){
                temp = temp + a_si[i] + b_si[i]*populationF[t][N_h+i] + c_si[i]*pow(populationF[t][N_h+i],2)
                + fabs(d_si[i]*sin(e_si[i]*(P_smin[i] - populationF[t][N_h+i])));
            }

        }
        F_fitness = temp;//default place for the fitness value
        return F_fitness;

}

void elitist(){//get all elite population and best individual
    population_elite.clear();
    if(best_fitness>population_fitness[0]){
        best_fitness = population_fitness[0];
    }
    for(int i=0;i<P_e;i++){
        population_elite.push_back(population[i]);
    }
}

void sbx(){
    double u;
    do{
        u = (double)rand()/rand();
        u-=(int)u;
    }while(isinf(u) || u!=u);
    for(int itr=0;itr<P_c;itr+=2){

        for(int t=0;t<T;t++){

            for(int i=0;i<N_h;i++){
                //compute for beta,gamma, u
                double x1;
                double x2;
                double min;
                if(population_crossover[itr][t][i]>=population_crossover[itr+1][t][i]){
                    x1 = population_crossover[itr][t][i];
                    x2 = population_crossover[itr+1][t][i];
                }
```

```
if(population_crossover[itr][t][i]<population_crossover[itr+1][t][i]){
    x1 = population_crossover[itr][t][i];
    x2 = population_crossover[itr+1][t][i];
}
if(population_crossover[itr][t][i]!=population_crossover[itr][t][i] || isinf(population_crossover[itr][t][i])){
    population_crossover[itr][t][i] = Q_hmax[i];
    i--;
    continue;
}
if(population_crossover[itr+1][t][i]!=population_crossover[itr+1][t][i] || isinf(population_crossover[itr+1][t][i])){
    population_crossover[itr+1][t][i] = Q_hmax[i];
    i--;
    continue;
}
if(x1==x2){
    continue;
}
else{
    if((x1-Q_hmin[i])<(x2-Q_hmax[i])){
        min = x1-Q_hmin[i];
    }
    else{
        min = x2-Q_hmax[i];
    }
    double beta = 1 + ((double)2/(x2-x1))*min;
        if(x2!=x2 || isinf(x2)){
            i--;
            continue;
        }
        if(x1!=x1 || isinf(x1)){
            i--;
            continue;
        }
    if(beta!=beta || isinf(beta)){
        i--;
        continue;
    }
    double asdf = -(n_c+1);
    double alpha;
    if(beta<0){
        alpha = 2 - -pow(fabs(beta),asdf);
    }
    else{
        alpha = 2 - pow(fabs(beta),asdf);
    }
    if(alpha!=alpha || isinf(alpha)){
        i--;
        continue;
    }
    double gamma = 0;
    if(u<=(1/alpha)){
        if((alpha*u)<0){
            gamma = -pow(fabs((alpha*u)),((double)1/(n_c+1)));
        }
        else{
            gamma = pow((alpha*u),((double)1/(n_c+1)));
        }
    }
    else{
        if((1/(2-alpha*u))<0){
            gamma = -pow(fabs(((double)1/(2-alpha*u))),((double)1/(n_c+1)));
        }
        else{
```

```
                gamma = pow(((double)1/(2-alpha*u)),((double)1/(n_c+1)));
            }
        }
        if(gamma!=gamma || isinf(gamma)){
            i--;
            continue;
        }
        double y1 = 0.5*((x1+x2) - gamma*fabs((x2-x1)));
        double y2 = 0.5*((x1+x2) + gamma*fabs((x2-x1)));
        population_crossover[itr][t][i]=y1;
        population_crossover[itr+1][t][i]=y2;
        if(population_crossover[itr][t][i]!=population_crossover[itr][t][i] || isinf(population_crossover[itr][t][i])){
            population_crossover[itr][t][i] = Q_hmax[i];
        }
        if(population_crossover[itr+1][t][i]!=population_crossover[itr+1][t][i] || isinf(population_crossover[itr+1][t][i])){
            population_crossover[itr+1][t][i] = Q_hmax[i];
        }
    }
}

for(int i=N_h;i<N_h+N_s;i++){
    double x1;
    double x2;
    double min;
    if(population_crossover[itr][t][i]>=population_crossover[itr+1][t][i]){
        x1 = population_crossover[itr][t][i];
        x2 = population_crossover[itr+1][t][i];
    }
    if(population_crossover[itr][t][i]<population_crossover[itr+1][t][i]){
        x1 = population_crossover[itr][t][i];
        x2 = population_crossover[itr+1][t][i];
    }
    if(population_crossover[itr][t][i]!=population_crossover[itr][t][i] || isinf(population_crossover[itr][t][i])){
        population_crossover[itr][t][i] = P_smin[i-N_h];
    }
    if(population_crossover[itr+1][t][i]!=population_crossover[itr+1][t][i] || isinf(population_crossover[itr+1][t][i])){
        population_crossover[itr+1][t][i] = P_smin[i-N_h];
    }
    if(x1==x2){
        continue;
    }
    else{
        if((x1-P_smin[i-N_h])<(x2-P_smax[i-N_h])){
            min = x1-P_smin[i-N_h];
        }
        else{
            min = x2-P_smax[i-N_h];
        }
        double beta = 1 + ((double)2/(x2-x1))*min;
        double asdf = -(n_c+1);
        double alpha;
        if(beta<0){
            alpha = 2- -pow(fabs(beta),asdf);
        }
        else{
            alpha = 2- pow(fabs(beta),asdf);
        }
        double gamma = 0;
        if(u<=(1/alpha)){
            if((alpha*u)<0){
                gamma = -pow(fabs((alpha*u)),((double)1/(n_c+1)));
            }
            else{
```

```
                        gamma = pow((alpha*u),((double)1/(n_c+1)));
                    }
                }
                else{
                    if((1/(2-alpha*u))<0){
                        gamma = -pow(fabs((double)(1/(2-alpha*u))),((double)1/(n_c+1)));
                    }
                    else{
                        gamma = pow(((double)1/(2-alpha*u)),((double)1/(n_c+1)));
                    }
                }
                double y1 = 0.5*((x1+x2) - gamma*fabs((x2-x1)));
                double y2 = 0.5*((x1+x2) + gamma*fabs((x2-x1)));
                population_crossover[itr][t][i]=y1;
                population_crossover[itr+1][t][i]=y2;
                if(population_crossover[itr][t][i]!=population_crossover[itr][t][i] || isinf(population_crossover[itr][t][i])){
                    population_crossover[itr][t][i] = P_smin[i-N_h];
                }
                if(population_crossover[itr+1][t][i]!=population_crossover[itr+1][t][i] || isinf(population_crossover[itr+1][t][i])){
                    population_crossover[itr+1][t][i] = P_smin[i-N_h];
                }
            }
        }
    }

    }
}

void crossover(){
    population_crossover.clear();
    deque<int> indexes;//list of indexes for crossover, if picked, pick another
    //pick crossover parents
    for(int i=P_e;i<N_p;i++){//index list for N_p-P_e
        indexes.push_back(i);
    }
    for(int i=0;i<P_c;i++){
        int itr1 = rand()%(N_p-P_e) - i;//itr1 ranges from P_e to N_p
        if(itr1<0) itr1=0;
        population_crossover.push_back(population[indexes[itr1]]);
        indexes.erase(indexes.begin()+(itr1));
    }
    //SBX
    sbx();
}

void mutation(){
    population_mutation.clear();
    deque<int> indexes;//list of indexes for crossover, if picked, pick another
    //pick crossover parents
    for(int i=P_e;i<N_p;i++){//index list for N_p-P_e
        indexes.push_back(i);
    }
    for(int i=0;i<P_m;i++){
        int itr1 = rand()%(N_p-P_e) - i;//itr1 ranges from P_e to N_p
        if(itr1<0) itr1=0;
        population_mutation.push_back(population[indexes[itr1]]);
        indexes.erase(indexes.begin()+(itr1));
    }

    for(int i=0;i<P_m;i++){
        for(int t=0;t<T;t++){
            for(int j=0;j<N_h;j++){
                bool teta = rand()%2;
```

```
    if(teta==0){
        double ra;
        do{
            ra = (double)rand()/rand();
            ra -= (int) ra;
        }while(isinf(ra) || ra!=ra);
        double temporary;
        if(((double)1-((double)t/T))<0){
            temporary = (double)t/T;
            temporary = 1-temporary;
            temporary = -pow(fabs(temporary), b_mutation);
            temporary = 1-pow(ra,temporary);
            population_mutation[i][t][j] = population_mutation[i][t][j] + (Q_hmax[j]-population_mutation[i][t][j]) * temporary;
        }
        else{
            temporary = (double)t/T;
            temporary = 1-temporary;
            temporary = pow(temporary, b_mutation);
            temporary = 1-pow(ra,temporary);
            population_mutation[i][t][j] = population_mutation[i][t][j] + (Q_hmax[j]-population_mutation[i][t][j]) * temporary;
        }

        if(population_mutation[i][t][j]!=population_mutation[i][t][j] || isinf(population_mutation[i][t][j])){
            population_mutation[i][t][j] = Q_hmax[j];
        }
    }
    if(teta==1){
        double ra;
        do{
            ra = (double)rand()/rand();
            ra -= (int) ra;
        }while(isinf(ra) || ra!=ra);

        double temporary;
        if(((double)1-((double)t/T))<0){
            temporary = (double)t/T;
            temporary = 1-temporary;
            temporary = -pow(fabs(temporary), b_mutation);
            temporary = 1-pow(ra,temporary);
            population_mutation[i][t][j] = population_mutation[i][t][j] + (population_mutation[i][t][j] - Q_hmin[j]) * temporary;
        }
        else{
            temporary = (double)t/T;
            temporary = 1-temporary;
            temporary = pow(temporary, b_mutation);
            temporary = 1-pow(ra,temporary);
            population_mutation[i][t][j] = population_mutation[i][t][j] + (population_mutation[i][t][j] - Q_hmin[j]) * temporary;
        }
        if(population_mutation[i][t][j]!=population_mutation[i][t][j] || isinf(population_mutation[i][t][j])){
            population_mutation[i][t][j] = Q_hmax[j];
        }
    }
}
for(int j=N_h;j<N_h+N_s;j++){
    bool teta = rand()%2;
    if(teta==0){
        double ra;
        do{
            ra = (double)rand()/rand();
            ra -= (int) ra;
        }while(isinf(ra) || ra!=ra);

        double temporary;
```

```
                    if(((double)1-((double)t/T))<0){
                        temporary = (double)t/T;
                        temporary = 1-temporary;
                        temporary = -pow(fabs(temporary), b_mutation);
                        temporary = 1-pow(ra,temporary);
                        population_mutation[i][t][j] = population_mutation[i][t][j] + (P_smax[i-N_h]-population_mutation[i][t][j]) * temporary;
                    }
                    else{
                        temporary = (double)t/T;
                        temporary = 1-temporary;
                        temporary = pow(temporary, b_mutation);
                        temporary = 1-pow(ra,temporary);
                        population_mutation[i][t][j] = population_mutation[i][t][j] + (P_smax[i-N_h]-population_mutation[i][t][j]) * temporary;
                    }
                    if(population_mutation[i][t][j]!=population_mutation[i][t][j] || isinf(population_mutation[i][t][j])){
                        population_mutation[i][t][j] = P_smin[i-N_h];
                    }
                }
                if(teta==1){
                    double ra;
                    do{
                        ra = (double)rand()/rand();
                        ra -= (int) ra;
                    }while(isinf(ra) || ra!=ra);
                    double temporary;
                    if(((double)1-((double)t/T))<0){
                        temporary = (double)t/T;
                        temporary = 1-temporary;
                        temporary = -pow(fabs(temporary), b_mutation);
                        temporary = 1-pow(ra,temporary);
                        population_mutation[i][t][j] = population_mutation[i][t][j] + (population_mutation[i][t][j]-P_smin[i-N_h]) * temporary;
                    }
                    else{
                        temporary = (double)t/T;
                        temporary = 1-temporary;
                        temporary = pow(temporary, b_mutation);
                        temporary = 1-pow(ra,temporary);
                        population_mutation[i][t][j] = population_mutation[i][t][j] + (population_mutation[i][t][j]-P_smin[i-N_h]) * temporary;
                    }
                    if(population_mutation[i][t][j]!=population_mutation[i][t][j] || isinf(population_mutation[i][t][j])){
                        population_mutation[i][t][j] = P_smin[i-N_h];
                    }
                }
            }
        }
    }
}

double transmission_loss(int i, int t){
    double temp1 = 0;
    double temp2 = 0;

    for(int itr1 = 0;itr1<N_h+N_s;itr1++){
        for(int itr2=0;itr2<N_h+N_s;itr2++){
            temp1 += population[i][t][itr1] * B[itr1][itr2] * population[i][t][itr2];
        }
    }

    for(int itr1=0;itr1<N_s+N_h;itr1++){
        temp2 += B0[itr1] * population[i][t][itr1];
    }
    double temp3 = temp1+temp2+B00;
    return temp3;
```

```
}

void update_population(){
    for(int i=0;i<P_e;i++){//add elite population to P
        population[i].swap(population_elite[i]);
    }
    for(int i=0;i<P_c;i++){//add crossover population to P
        population[P_e+i].swap(population_crossover[i]);
    }
    for(int i=0;i<P_m;i++){
        population[i+P_e+P_c].swap(population_mutation[i]);
    }

}

double euclidian(deque<deque<double> > temp1, deque<deque<double> > temp2){
    double temp=0;
    for(int t=0;t<T;t++){
        for(int j=0;j<N_h+N_s;j++){
            double tempVar = (temp1[t][j] - temp2[t][j]);
                temp = pow(fabs(tempVar),2);
                if(temp!=temp){
                }

        }
    }
    return sqrt(temp);//return always positive due to square power
}

double evaluation(deque<deque<double> > temp1){
    double temp=0;
    for(int t=0;t<T;t++){
        for(int i=N_h;i<N_h+N_s;i++){
            temp = temp + a_si[i] + b_si[i]*temp1[t][N_h+i] + c_si[i]*pow(temp1[t][N_h+i],2) + fabs(d_si[i]*sin(e_si[i]*(P_smin[i] - temp1[t][N_h+i])));
        }
    }
    return temp;
}

double preyS(int index){
    for(int ind = 0;ind<N_p;ind++){
        if(board[index][ind]==1){
            if( evaluation(population[ind])<S_fitness){//random fish is better
                for(int t=0;t<T;t++){
                    for(int j=0;j<N_h+N_s;j++){
                        double ran;
                        do{
                            ran = (double)rand()/rand();
                            ran -=(int) ran;
                        }while(isinf(ran) || ran!=ran);

                        populationS[t][j] = populationS[t][j] + ( ((double)(population[ind][t][j] - populationS[t][j])/euclidian(population[ind],populationS))
                        *step*ran);
                    }
                }
            }
            else{//random fish is not better, perform free move
                for(int t=0;t<T;t++){
                    for(int j=0;j<N_h+N_s;j++){
                        double ran = (double)rand()/rand();
                        ran -= (int) ran;
                        if(isinf(ran) || ran!=ran){
                            j--;
```

```
                        continue;
                    }
                    populationS[t][j] = populationS[t][j] + step*ran;
                }
            }

        }
        return evaluationS();
    }
}
}

double preyF(int index){
    for(int ind=0;ind<N_p;ind++){
        if(board[index][ind]==1){
            if( evaluation(population[ind])<F_fitness){//random fish is better
                for(int t=0;t<T;t++){
                    for(int j=0;j<N_h+N_s;j++){
                        double ran = (double)rand()/rand();
                        ran -=(int) ran;
                        if(isinf(ran) || ran!=ran){
                            j--;
                            continue;
                        }
                        populationF[t][j] = populationF[t][j] + ( ((double)(population[ind][t][j] - populationF[t][j])/euclidian(population[ind],populationF))
                        *step*ran);
                    }
                }
            }
            else{//random fish is not better, perform free move
                for(int t=0;t<T;t++){
                    for(int j=0;j<N_h+N_s;j++){
                        double ran = (double)rand()/rand();
                        ran -= (int) ran;
                        if(isinf(ran) || ran!=ran){
                            j--;
                            continue;
                        }
                        populationF[t][j] = populationF[t][j] + step*ran;
                    }
                }
            }
            return evaluationF();
        }

    }
}

double swarm(int index){
    deque<deque<double> > center;
    populationS.clear();
    populationS = population[index];
    S_fitness = population_fitness[index];
    for(int t=0;t<T;t++){
        deque<double> a;
        for(int j=0;j<N_h+N_s;j++){
            a.push_back(0);
        }
        center.push_back(a);
        a.clear();
    }

    double N_visual=0;
```

```
        for(int itr1=0;itr1<POP_SIZE;itr1++){
            if(board[index][itr1]==1){
                for(int t=0;t<T;t++){
                    for(int j=0;j<N_h+N_s;j++){
                        center[t][j] += population[itr1][t][j];
                    }
                }
                N_visual++;
            }
        }

        if(N_visual==0){
            return preyS(index);
        }
        else{
            for(int t=0;t<T;t++){//divide all center chromosomes to N_visual
                for(int j=0;j<N_h+N_s;j++){
                    center[t][j] = (double)center[t][j]/N_visual;
                    if(center[t][j]!=center[t][j]){
                    }
                }
            }
            //perform swarm behavior
            double euclid = euclidian(populationS,center);
            if(evaluation(center)<(population_fitness[index]*crowd_factor)){//center is better
                for(int t=0;t<T;t++){//apply swarm function
                    for(int j=0;j<N_s+N_h;j++){
                        double ran = (double)rand()/rand();
                        ran -=(int)ran;
                        if(isinf(ran) || ran!=ran){
                            j--;
                            continue;
                        }
                        populationS[t][j] = populationS[t][j] + ((double)(center[t][j]-populationS[t][j])/euclid)*step*ran;
                    }
                }
                return evaluationS();
            }
            else{//current position is better, perform prey behavior
                return preyS(index);
            }
        }
}

double follow(int index){
    populationF = population[index];
    for(int itr1=0;itr1<POP_SIZE;itr1++){//first in deque is already the best, just find the first best visible fish
        if(board[index][itr1]==1){//itr1 fish is visible
            if(itr1>index){//index fish is already the best fish, no follow behavior needed
                return preyF(index);//follow not possible, do prey behavior
            }
            else{//perform follow behavior
                for(int t=0;t<T;t++){
                    for(int j=0;j<N_h+N_s;j++){
                        double ran = (double)rand()/rand();
                        ran -= (int)ran;
                        if(isinf(ran)|| ran!=ran){
                            j--;
                            continue;
                        }
                        populationF[t][j] = populationF[t][j] + (((double)(population[itr1][t][j] - populationF[t][j])/euclidian(populationF,population[itr1]))
                        *step*ran);
                        if(populationF[t][j]!=populationF[t][j] || isinf(populationF[t][j])){
```

```
                    }
                }
            }
            return evaluationF();
        }
    }
  }
}

void blackboard(){
    //reset
    for(int itr1=0;itr1<POP_SIZE;itr1++){
        for(int itr2=0;itr2<POP_SIZE;itr2++){
            if(itr1==itr2){
                board[itr1][itr2] = 0;
            }
            else{
                board[itr1][itr2]= -1;
            }
        }
    }
    //get visible fishes
    for(int itr1=0;itr1<POP_SIZE;itr1++){
        for(int itr2=0;itr2<POP_SIZE;itr2++){
            if(itr1==itr2){
                continue;
            }
            else{
                if(euclidian(population[itr1],population[itr2])<=visual){
                    board[itr1][itr2] = 1;
                }
            }
        }
    }
}

void evaluation_pso(){//calculate fitness value for population
    //cout<<"evaluation_pso"<<endl;
    population_pso_fitness.clear();
    for(int itr=0;itr<POP_SIZE;itr++){
        double temp=0;
        for(int t=0;t<T;t++){
            for(int i=0;i<N_s;i++){
                //cout<<itr<<" "<<t<<" "<<i<<endl;
                temp = temp + a_si[i] + b_si[i]*population_pso[itr][t][N_h+i] + c_si[i]*pow(population_pso[itr][t][N_h+i],2)
                + fabs(d_si[i]*sin(e_si[i]*(P_smin[i] - population[itr][t][N_h+i])));
                //cout<<temp<<" ";
            }
            //cout<<endl;

        }

        population_pso_fitness.push_back(temp);//default place for the fitness value
    }
}

void initialize_pso(){//initialize all velocity to 0
    for(int i=0;i<N_p;i++){
        deque<deque<double> > temp1;
        for(int t=0;t<T;t++){
            deque<double> temp2;

            for(int j=0;j<N_h;j++){
```

```
                bool ran = rand()%2;
                double temp3 = abs(static_cast<double> (rand() / static_cast<double> (RAND_MAX/(Q_hmin[j] - Q_hmax[j]) )));
                if(ran==true){
                    temp3 = -temp3;
                }
                if(isinf(temp3) || temp3!=temp3){
                    j--;
                    continue;
                }
                temp2.push_back(temp3);
            }

            for(int j=N_h;j<N_h+N_s;j++){
                bool ran = rand()%2;
                double temp3 = abs(static_cast<double> (rand() / static_cast<double> (RAND_MAX/(Q_hmin[j] - Q_hmax[j]) )));
                if(ran==true){
                    temp3 = -temp3;
                }
                if(isinf(temp3) || temp3!=temp3){
                    j--;
                    continue;
                }
                temp2.push_back(temp3);
            }
            temp1.push_back(temp2);
            temp2.clear();
        }
            velocity.push_back(temp1);
            temp1.clear();
    }

    for(int i=0;i<N_p;i++){//copy population to pbest
        deque<deque<double> > temp1;
        for(int t=0;t<T;t++){
            deque<double> temp2;
            for(int j=0;j<N_h+N_s;j++){
                temp2.push_back(population[i][t][j]);
            }
            temp1.push_back(temp2);
            temp2.clear();
        }
        pbest.push_back(temp1);
        temp1.clear();
        pbest_fitness.push_back(population_fitness[i]);
    }
        //initialize population_pso to zeros
    for(int i=0;i<N_p;i++){
        deque<deque<double> > temp1;
        for(int t=0;t<T;t++){
            deque<double> temp2;
            for(int j=0;j<N_h+N_s;j++){
                temp2.push_back(0);
            }
            temp1.push_back(temp2);
            temp2.clear();
        }
        population_pso.push_back(temp1);
        temp1.clear();
    }
        //get first as gbest
        gbest = population[0];
        gbest_fitness = population_fitness[0];
}
```

```
double update_velocity(int i, int t, int j, int itr){//returns updated velocity of i t j
        double rp, rg;
        double varphi_p = 2.05;
        double varphi_g = 2.1;
        double C;
        double w;
        double wmin = 0.1;
        double wmax = 1.0;
        double varphi = varphi_g+varphi_g;
        C = 2/(2-varphi - sqrt(pow(varphi,2) - 4*varphi ));
        w = (double)(wmax-wmin)*((double)(gmax_pso - itr)/gmax_pso) + wmin;
        do{
            rp = (double)rand()/rand();
            rp -= (int)rp;
        }while(isinf(rp) || rp!=rp);
        do{
            rg = (double)rand()/rand();
            rg -= (int)rg;
        }while(isinf(rg) || rg!=rg);

        double newv = (C*(w*velocity[i][t][j] + rp*varphi_p*(pbest[i][t][j] - population[i][t][j]) + rg*varphi_g*(gbest[t][j] - population[i][t][j]) ) );
        return newv;
}

void pso(int itr){//run pso for all population
    for(int i=0;i<N_p;i++){
        for(int t=0;t<T;t++){
            for(int j=0;j<N_h+N_s;j++){
                velocity[i][t][j] = update_velocity(i,t,j,itr);
                population_pso[i][t][j] = population[i][t][j] + velocity[i][t][j];
            }
        }
    }
    //evaluation
    evaluation_pso();

    //update pbest
    for(int i=0;i<N_p;i++){
        if(population_fitness[i]<pbest_fitness[i]){
            for(int t=0;t<T;t++){
                for(int j=0;j<N_h+N_s;j++){
                    pbest[i][t][j] = population[i][t][j];
                }
            }
            pbest_fitness[i] = population_fitness[i];
        }
        else{
            continue;
        }
    }

    //update population if necessary from population_pso
    for(int i=0;i<N_p;i++){
        if(population_pso_fitness[i] < population_fitness[i]){
            population[i].swap(population_pso[i]);
            population_fitness[i] = population_pso_fitness[i];
        }
    }


    //update gbest
    double bestIndex;
```

```
    for(int i=0;i<N_p;i++){
        double temp = 999999999999999;
        if(population_fitness[i]<temp){
            temp = population_fitness[i];
            bestIndex = i;
        }
    }
    gbest_fitness = population_fitness[bestIndex];
    for(int t=0;t<T;t++){
        for(int j=0;j<N_h+N_s;j++){
            gbest[t][j] = population[bestIndex][t][j];
        }
    }
}

double update_velocity1(int i, int t, int j, int itr){//returns updated velocity of i t j
        double rp, rg;
        double varphi_p = 2.05;
        double varphi_g = 2.1;
        double C;
        double w;
        double wmin = 0.1;
        double wmax = 1.0;
        double varphi = varphi_g+varphi_g;
        C = 2/(2-varphi - sqrt(pow(varphi,2) - 4*varphi ));
        w = (double)(wmax-wmin)*((double)(gmax_pso - itr)/gmax_pso) + wmin;
        do{
            rp = (double)rand()/rand();
            rp -= (int)rp;
        }while(isinf(rp) || rp!=rp);
        do{
            rg = (double)rand()/rand();
            rg -= (int)rg;
        }while(isinf(rg) || rg!=rg);
        double newv = (C*(w*velocity[i][t][j] + rp*varphi_p*(pbest[i][t][j] - population[i][t][j]) + rg*varphi_g*(gbest[t][j] - population[i][t][j]) ) );
        return newv;
}

void sohpso(int itr){//run pso for all population
    for(int i=0;i<N_p;i++){
        for(int t=0;t<T;t++){
            for(int j=0;j<N_h+N_s;j++){
                velocity[i][t][j] = update_velocity1(i,t,j,itr);
                population_pso[i][t][j] = population[i][t][j] + velocity[i][t][j];
            }
        }
    }
    //evaluation
    evaluation_pso();
    //update population if necessary from population_pso
    for(int i=0;i<N_p;i++){
        if(population_pso_fitness[i] < population_fitness[i]){
            population[i].swap(population_pso[i]);
            population_fitness[i] = population_pso_fitness[i];
        }
    }
    //sort all individuals
    for(int itr1=0;itr1<POP_SIZE;itr1++){
        for(int itr2=0;itr2<POP_SIZE;itr2++){
            if(population_fitness[itr1]<population_fitness[itr2]){
                population[itr1].swap(population[itr2]);
                double huehue = population_fitness[itr1];
                population_fitness[itr1] = population_fitness[itr2];
```

```
                population_fitness[itr2] = huehue;
            }
        }
    }

    if(population_fitness[0] < gbest_fitness){
        gbest = population[0];
        gbest_fitness = population_fitness[0];
    }
}


void constraints_handling(){
    for(int i=0;i<POP_SIZE;i++){
        for(int j=0;j<N_h;j++){//constraints handling for hydro plants
            int iteration;
            iteration = 0;
            check_discharge(i,j);//check Q_h(j,t)
            double V_hc = volume[i][T-1][j] - V_hend[j];
            while(fabs(V_hc)>e_vcourse && iteration<=iteration_course){
                double V_haverage = V_hc/T;
                for(int t=0;t<T;t++){
                    population[i][t][j] = population[i][t][j] + V_haverage;
                    if(population[i][t][j]!=population[i][t][j] || isinf(population[i][t][j])){
                        population[i][t][j] = Q_hmax[j];
                    }
                    check_discharge1(i,j,t);
                }
                update_volume(i);
                V_hc = volume[i][T-1][j] - V_hend[j];
                iteration++;
            }
            iteration=0;
            deque<double> Q_hc;
            while(fabs(V_hc)>e_vfine && iteration<iteration_fine){
                Q_hc.clear();
                Q_hc = Q_hcF(i,j);//get the change in water discharge population from i and j
                int itr = max_adjustableQ(Q_hc);
                population[i][itr][j] = population[i][itr][j] + V_hc;
                if(population[i][itr][j]!=population[i][itr][j] || isinf(population[i][itr][j])){
                    population[i][itr][j] = Q_hmax[j];
                }
                update_volume(i);
                check_discharge1(i,j,itr);
                V_hc = volume[i][T-1][j] - V_hend[j];
                iteration++;
            }
        }
        update_volume(i);

        for(int t=0;t<T;t++){//thermal plants constraints handling
            int iteration = 0;
            double P_sc = P_D[t] + transmission_loss(i,t) - P_ssum(i,t) - P_hsum(i,t);//no transmission loss
            while(fabs(P_sc)>e_pcourse && iteration<=iteration_course){
                double P_taverage = P_sc/N_s;

                for(int ns = N_h; ns<N_h+N_s;ns++){
                    population[i][t][ns] = population[i][t][ns] + P_taverage;
                    if(population[i][t][ns]!=population[i][t][ns] || isinf(population[i][t][ns])){
                        population[i][t][ns] = P_smin[ns-N_h];
                    }
                    check_thermal1(i,t,ns);
                }
```

```
                P_sc = P_D[t] + transmission_loss(i,t) - P_ssum(i,t) - P_hsum(i,t);//no transmission loss
                iteration++;
            }
            iteration =0;
            while(fabs(P_sc)>e_pfine && iteration<=iteration_fine){
                P_sc = P_D[t] + transmission_loss(i,t) - P_ssum(i,t) - P_hsum(i,t);
                int itr = max_adjustableP(i,t);
                population[i][t][itr] = population[i][t][itr] + P_sc;
                if(population[i][t][itr]!=population[i][t][itr] || isinf(population[i][t][itr])){
                    population[i][t][itr] = P_smin[itr-N_h];
                }
                P_sc = P_D[t] + transmission_loss(i,t) - P_ssum(i,t) - P_hsum(i,t);
                iteration++;
            }
        }
    }
    for(int i=0;i<POP_SIZE;i++){
        for(int t=0;t<T;t++){
            for(int j=0;j<N_h;j++){
                if(isinf(population[i][t][j]) || (population[i][t][j]!=population[i][t][j])){
                    population[i][t][j] = Q_hmax[j];
                }
            }
            for(int j=N_h;j<N_h+N_s;j++){
                if(isinf(population[i][t][j]) || (population[i][t][j]!=population[i][t][j])){
                    population[i][t][j] = P_smin[j-N_h];
                }
            }
        }
    }
}

int main(){
best_fitness = 999999999999999;
inflowF();
initUpstream();
aveTime = 0;
for(int t=0;t<T;t++){//initialization of super global variable
    deque<double> temps;//chromosome
    deque<double> temps1;//Ph Qh
    deque<double> temps2;//Ps
    if(t==0){
        for(int j=0;j<N_h+N_s;j++){
            temps.push_back(0);
        }
        for(int j=0;j<N_h;j++){
            temps1.push_back(0);
        }
        for(int j=N_h;j<N_h+N_s;j++){
            temps2.push_back(0);
        }
    }
    aveBestChromosome.push_back(temps);
    aveWorstChromosome.push_back(temps);
    aveBestPh.push_back(temps1);
    aveBestPs.push_back(temps2);
    aveBestQh.push_back(temps1);
}



for(int mainItr=0;mainItr<iterators;mainItr++){
    population.clear();
```

```
population_fitness.clear();
volume.clear();
population_elite.clear();
population_crossover.clear();
population_mutation.clear();
populationS.clear();
populationF.clear();
S_fitness = 0;
F_fitness = 0;
best_fitness = 9999999999999999;
srand(time(NULL));
//INITIALIZATION
for(int i=0;i<POP_SIZE;i++){//temp(n), n - dimension
    deque<deque<double> > temp2;
    for(int t=0;t<T;t++){
        deque<double> temp1;

        for(int nh=0;nh<N_h;nh++){
            double ran;
            double temp;
            do{
                ran = (double)rand()/rand();
                ran -= (int) ran;
            }while(isinf(ran) || ran!=ran);

            temp = Q_hmin[nh] + ran*(Q_hmax[nh]-Q_hmin[nh]);
            temp1.push_back(temp);
        }

        for(int ns=N_h;ns<N_h+N_s;ns++){
            double ran;
            double temp;
            do{
                ran = (double)rand()/rand();
                ran -= (int) ran;
            }while(isinf(ran) || ran!=ran);
            temp = P_smin[ns-N_h] + ran*(P_smax[ns-N_h]-P_smin[ns-N_h]);
            temp1.push_back(temp);
        } temp2.push_back(temp1);
        temp1.clear();
    }
    population.push_back(temp2);
    temp2.clear();
}

for(int i=0;i<POP_SIZE;i++){//water dynamic volume generation
    //water dynamic balance generation
    deque<deque<double> > V_h;//V_h(t,j)
    V_h.clear();
    for(int t=0;t<T;t++){
        deque<double> temp1;
        for(int j=0;j<N_h;j++){
            double temp=0;//V_h(j)
            if((t-1)<0){
                temp = V_hbegin[j];// + inflow[j][t] - population[i][t][j] + sum_upstream(i,t,j);//spillage not calculated
                temp1.push_back(temp);
            }
            else{
                temp = V_h[t-1][j] + inflow[j][t] - population[i][t][j] + sum_upstream(i,t,j);//spillage not calculated
                temp1.push_back(temp);
            }
        }
        V_h.push_back(temp1);
```

```
        temp1.clear();
    }
    volume.push_back(V_h);//always update volume reservoir after change in values
    V_h.clear();
}

//CONSTRAINTS HANDLING
for(int i=0;i<POP_SIZE;i++){
    for(int j=0;j<N_h;j++){//constraints handling for hydro plants
        int iteration;
        iteration = 0;
        check_discharge(i,j);//check Q_h(j,t)
        double V_hc = volume[i][T-1][j] - V_hend[j];
        while(fabs(V_hc)>e_vcourse && iteration<iteration_course){
            double V_haverage = V_hc/T;
            for(int t=0;t<T;t++){
                population[i][t][j] = population[i][t][j] + V_haverage;
                check_discharge1(i,j,t);
            }
            update_volume(i);
            V_hc = volume[i][T-1][j] - V_hend[j];
            iteration++;
        }
        iteration=0;
        deque<double> Q_hc;
        while(fabs(V_hc)>e_vfine && iteration<iteration_fine){
            Q_hc.clear();
            Q_hc = Q_hcF(i,j);//get the population from i and j, change in Q
            int itr = max_adjustableQ(Q_hc);
            population[i][itr][j] = population[i][itr][j] + V_hc;
            update_volume(i);
            check_discharge1(i,j,itr);
            V_hc = volume[i][T-1][j] - V_hend[j];
            iteration++;
        }
    }
    update_volume(i);

    for(int t=0;t<T;t++){//thermal plants constraints handling
        int iteration = 0;
        double P_sc = P_D[t] + transmission_loss(i,t) - P_ssum(i,t) - P_hsum(i,t);//no transmission loss
        while(fabs(P_sc)>e_pcourse && iteration<=iteration_course){
            double P_taverage = P_sc/N_s;

            for(int ns = N_h; ns<N_s;ns++){
                population[i][t][ns] = population[i][t][ns] + P_taverage;
                check_thermal1(i,t,ns);
            }
            P_sc = P_D[t] + transmission_loss(i,t) - P_ssum(i,t) - P_hsum(i,t);//no transmission loss
            iteration++;
        }
        iteration =0;
        while(fabs(P_sc)>e_pfine && iteration<=iteration_fine){
            P_sc = P_D[t] + transmission_loss(i,t) - P_ssum(i,t) - P_hsum(i,t);//violation in the equality constraints of power generated to power demand
            int itr = max_adjustableP(i,t);
            population[i][t][itr] = population[i][t][itr] + P_sc;
            P_sc = P_D[t] + transmission_loss(i,t) - P_ssum(i,t) - P_hsum(i,t);
            iteration++;
        }
    }
}
clock_t t;
t = clock();
```

```
//RCGA
for(int g = 0;g<gmax_rcga;g++){
    //EVALUATION
    evaluation();//generate fitness value and sort them

    elitist();//copy all the elite population, get best_fitness

    //CROSSOVER
    crossover();//pure crossover w/ sbx.

    //MUTATION
    mutation();//pure mutation

    //cout<<"MUTATION"<<endl;
    update_population();//add the crossover, mutated population to the general population

    constraints_handling();

    population_fitness.clear();
    population_elite.clear();
    population_crossover.clear();
    population_mutation.clear();
}
evaluation();
constraints_handling();

//AFSA
for(int g = 0;g<gmax_afsa;g++){
    evaluation();
    for(int i=0;i<POP_SIZE;i++){//perform evaluation and ranking after every functions and method
        best_fitness = population_fitness[0];
        double swarmVar = swarm(i);//swarmVar is the fitness value of ith fish that performed swarm
        double followVar = follow(i);//followVar is the fitness value of ith fish that performed follow
        if(swarmVar>followVar){//choose follow behavior
            population[i].swap(populationF);
            population_fitness[i] = F_fitness;
        }
        else{//choose swarm behavior
            population[i].swap(populationS);
            population_fitness[i] = S_fitness;
        }
    }
    constraints_handling();
    blackboard();
}

//PSO
initialize_pso();
for(int g = 0;g<gmax_pso;g++){
    pso(g);
    //constraints handling for pso
    constraints_handling();
}
//SOHPSO
initialize_pso();
for(int g = 0;g<gmax_sohpso;g++){
    sohpso(g);
    //constraints handling for pso
    constraints_handling();
}
t=clock() - t;

evaluation();
```

```
cout<<"----------------------------------------------------------------"<<endl;
aveBestFitness.push_back(population_fitness[0]);
aveWorstFitness.push_back(population_fitness[99]);

cout<<"Best Fitness: "<<population_fitness[0]<<endl;
cout<<"Worst Fitness: "<<population_fitness[99]<<endl;

for(int t=0;t<T;t++){
    for(int j=0;j<N_h+N_s;j++){
        aveBestChromosome[t][j]+=population[0][t][j];
    }
}

for(int t=0;t<T;t++){
    for(int j=0;j<N_h+N_s;j++){
        aveWorstChromosome[t][j]+=population[99][t][j];
    }
}
cout<<"----------------------------------------------------------------"<<endl;
cout<<"P_h: "<<endl;
for(int t=0;t<T;t++){//add average best P_h
    for(int a=0;a<N_h;a++){
        double temp = 0;
        temp = c1[a]*pow(volume[0][t][a],2) + c2[a]*pow(population[0][t][a],2) + c3[a]*volume[0][t][a]*population[0][t][a]
        + c4[a]*volume[0][t][a] + c5[a]*population[0][t][a] + c6[a];
        if(temp<0){
            temp=0;
        }cout<<temp<<" ";
        aveBestPh[t][a] += temp;
    }
    cout<<endl;
}
cout<<"----------------------------------------------------------------"<<endl;
for(int t=0;t<T;t++){
    for(int j=N_h;j<N_h+N_s;j++){
        aveBestPs[t][j] += population[0][t][j];
    }
}

for(int t=0;t<T;t++){
    for(int j=0;j<N_h;j++){
        aveBestQh[t][j] += population[0][t][j];
    }
}

cout<<"Best: "<<endl;
for(int i=0;i<1;i++){
    for(int t=0;t<T;t++){
        for(int j=0;j<N_h+N_s;j++){
            cout<<population[i][t][j]<<" ";
        }
        cout<<endl;
    }
    cout<<endl;
}
cout<<endl;


cout<<"Volume: "<<endl;
for(int i=0;i<2;i++){
    for(int t=0;t<T;t++){
        for(int j=0;j<N_h;j++){
            cout<<volume[i][t][j]<<" ";
```

```
            }
            cout<<endl;
        }
        cout<<endl;
    }
    cout<<endl;
    cout<<"----------------------------------------------------------------"<<endl;
    aveTime += (float)t/CLOCKS_PER_SEC;
    cout<<(float)t/CLOCKS_PER_SEC<<" secs"<<endl;
}//end of main iteration (30 runs)


for(int t=0;t<T;t++){
    for(int j=0;j<N_h+N_s;j++){
        aveBestChromosome[t][j] = (double)aveBestChromosome[t][j]/iterators;
        aveWorstChromosome[t][j] = (double)aveWorstChromosome[t][j]/iterators;
    }
}

for(int t=0;t<T;t++){
    for(int j=0;j<N_h;j++){
        aveBestPh[t][j] = (double)aveBestPh[t][j]/iterators;
        aveBestQh[t][j] = (double)aveBestQh[t][j]/iterators;
    }
}

for(int t=0;t<T;t++){
    for(int j=0;j<N_s;j++){
        aveBestPs[t][j] = (double)aveBestPs[t][j]/iterators;
    }
}

double aveBestF=0;
double aveWorstF=0;
for(int i=0;i<iterators;i++){
    aveBestF += aveBestFitness[i];
    aveWorstF += aveWorstFitness[i];
}
aveBestF = (double)aveBestF/iterators;
aveWorstF = (double)aveWorstF/iterators;

cout<<"Average Best: "<<aveBestF<<endl;
cout<<"Average Worst: "<<aveWorstF<<endl;
cout<<endl;
cout<<"============================================================"<<endl;
cout<<"Average Best Chromosome: "<<endl;
for(int t=0;t<T;t++){
    for(int j=0;j<N_h+N_s;j++){
        cout<<aveBestChromosome[t][j]<<" ";
    }
    cout<<endl;
}
cout<<"============================================================"<<endl;
cout<<"Average Worst Chromosome: "<<endl;
for(int t=0;t<T;t++){
    for(int j=0;j<N_h+N_s;j++){
        cout<<aveWorstChromosome[t][j]<<" ";
    }
    cout<<endl;
}
cout<<"============================================================"<<endl;
cout<<"Average Best P_h: "<<endl;
for(int t=0;t<T;t++){
```

```
    for(int j=0;j<N_h;j++){
        cout<<aveBestPh[t][j]<<" ";
    }
    cout<<endl;
}
cout<<"============================================================"<<endl;
cout<<"Average Best P_s"<<endl;
for(int t=0;t<T;t++){
    for(int j=0;j<N_s;j++){
        cout<<aveBestPs[t][j]<<" ";
    }
    cout<<endl;
}
cout<<"============================================================"<<endl;
cout<<"Total Generation:"<<endl;
for(int t=0;t<T;t++){
    double temp = 0;
    for(int j=0;j<N_h;j++){
        temp += aveBestPh[t][j];
    }
    for(int j=N_h;j<N_h+N_s;j++){
        temp += aveBestChromosome[t][j];
    }
    cout<<temp<<endl;
}
cout<<"============================================================"<<endl;
cout<<"Average Best Q_h"<<endl;
for(int t=0;t<T;t++){
    for(int j=0;j<N_h;j++){
        cout<<aveBestQh[t][j]<<" ";
    }
    cout<<endl;
}
cout<<"============================================================"<<endl;




return 0;
}
```