Assignment 2 – Symbolic Regression

Hongbo Zhu UNI: hz2629

Evolutionary Computation & Design Automation

(MECS E4510)

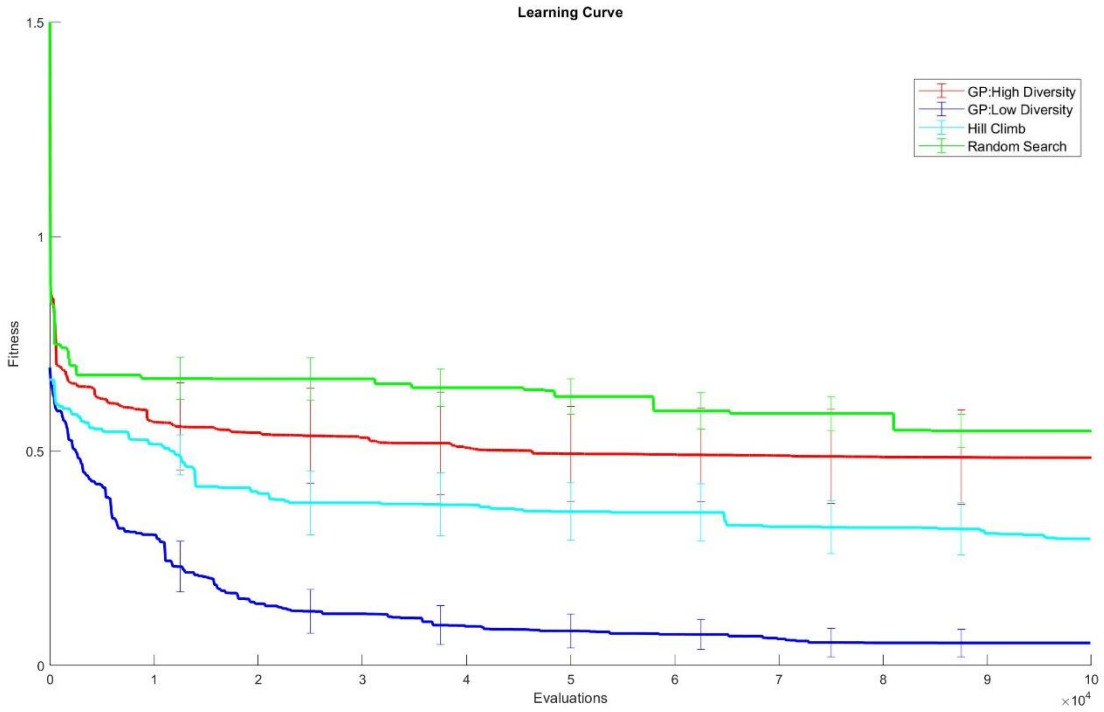
Instructor: Prof. Hod Lipson

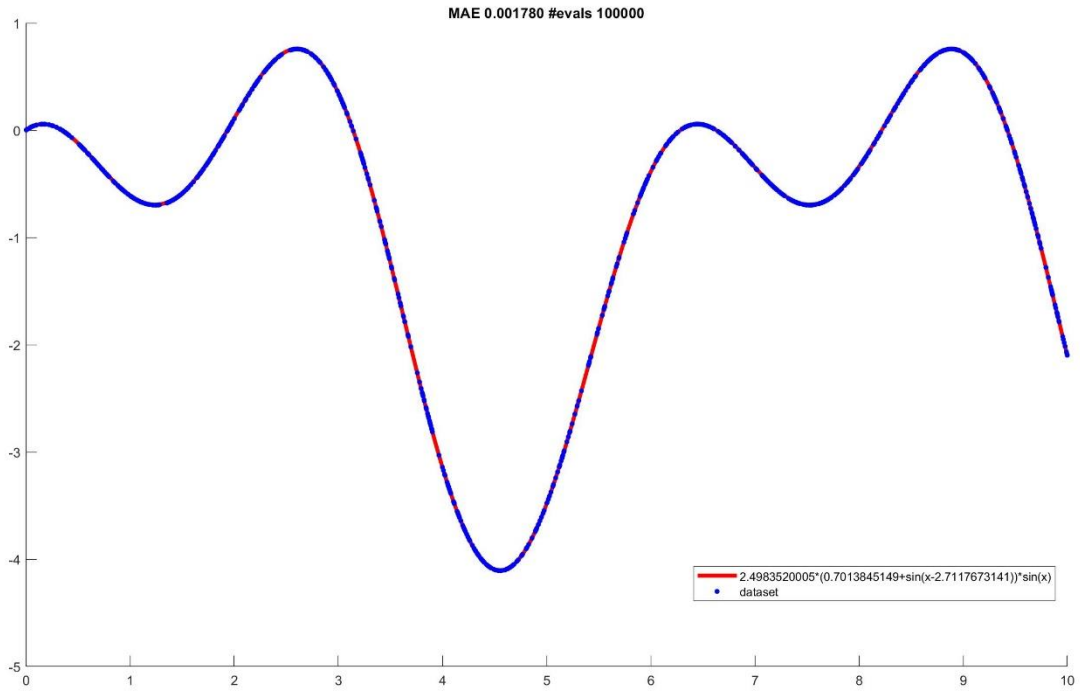
Date Submitted: 10/20/2019

Grace Hours Used: 0 h

Grace Hours Remaining: 102.5 h

1. Results summary table





2. Methods

2.1 Random Search

Random Search is used in this assignment to compare with genetic programming and hill climber method. This method generates a random tree each loop which contains operators, constants and variable in function, i.e. a list of gene, so that a random solution is generated each generation. By keeping the best solution's fitness, then replacing it with a new one if the fitness is better. After enough times generation, some random solutions are able to find a better fitness. This method is easy to program. However, it is not efficient, since each generation is separated so there is no useful part of tree remain after processing. Thus, this method is not a propriate way to find a solution of a complex problem which has a great searching space.

2.2 Hill Climber

Hill Climber is an efficient way to find an optimal solution in some problem. For Symbolic Regression problem, a random tree, i.e. an array contains operators and variables and constants to form a function, will be generated without repeat. Each node in a tree contains a part of function. Then this array will be processed in every generations. Each time, there are some mutations take place in this tree. The operators are probably be replaced by other kind of operators, like "+", "-", "*", "/", "sin", "cos", or multiplying the brunch of tree by a factor close to 1, adding a random number close to 0 to a branch, or even replace all branch with a random number. Besides, coefficients in function might be changed in small scale. If these operations improve the fitness, then this exchange, or mutation, will be kept. Vice versa, the mutation which does not improve the fitness will be abandon. By this way, mutation is positive to this gene. Compared with random search, useful information is kept and passed to the next the generation so that after many generation's accumulation, this algorithm is able to find a solution in searching space. However, this method is likely to get a local optimal solution instead of global optimal solution for some complex problem, since only mutation one gene once each time is not enough to find a better solution when the algorithm's outcome is a local optimal solution.

2.3 Presentation

In this assignment, a list of string is used to represent the function. This list is in the order of tree. This tree has less than 8 layers, and the index of a node in a tree is from 1 to 256, i.e. 2^8. If the index of a node of this tree in the list is "i", then the next two nodes are "2 * i" and "2 * i + 1". This presentation method reduces the time of searching a number or a string in the list, so it is useful for symbolic regression problem.

2.4 Selection Processes

Roulette Selection method and Truncation Selection method are implemented in this assignment. Both methods are ways to select certain individuals in population and remove the rest depend on individuals' fitness. When using Roulette Selection method, the probability of being selected is proportional to the fitness. By this method, high-fitness genes are more likely being selected to produce offspring in later processes. The other method, Truncation Selection method, sets an integer n before selecting, then ranks the individuals by their fitness. After ranking, top $1/n$ individuals are kept while others are eliminated. This method protects high-fitness gene so that the average fitness is able to grow.

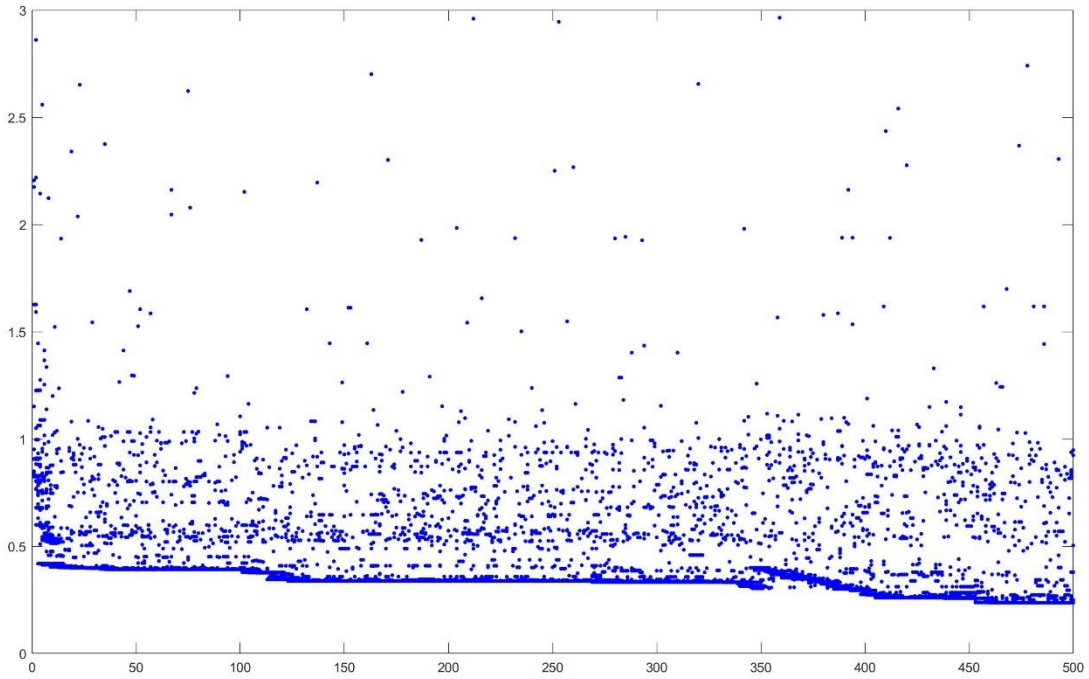
However, in the method using truncation selection, Hierarchical Fair Competition is used. In this assignment 100 individuals are in population group. Before the selection, worst 20 individuals are abandoned and replaced by 20 new random individuals. Then these 100 genes are ranked from best to worst. Then this population is divided in 5 group due to their fitness, which means 20 best individuals are in one group 20 worst ones are in last group and so on. After that, truncation selection is applied in each group separately so that new individuals are not been eliminated by old ones. Y using this method new genes are generated continuously so the whole population are able to have high diversity.

2.5 Variation Operators

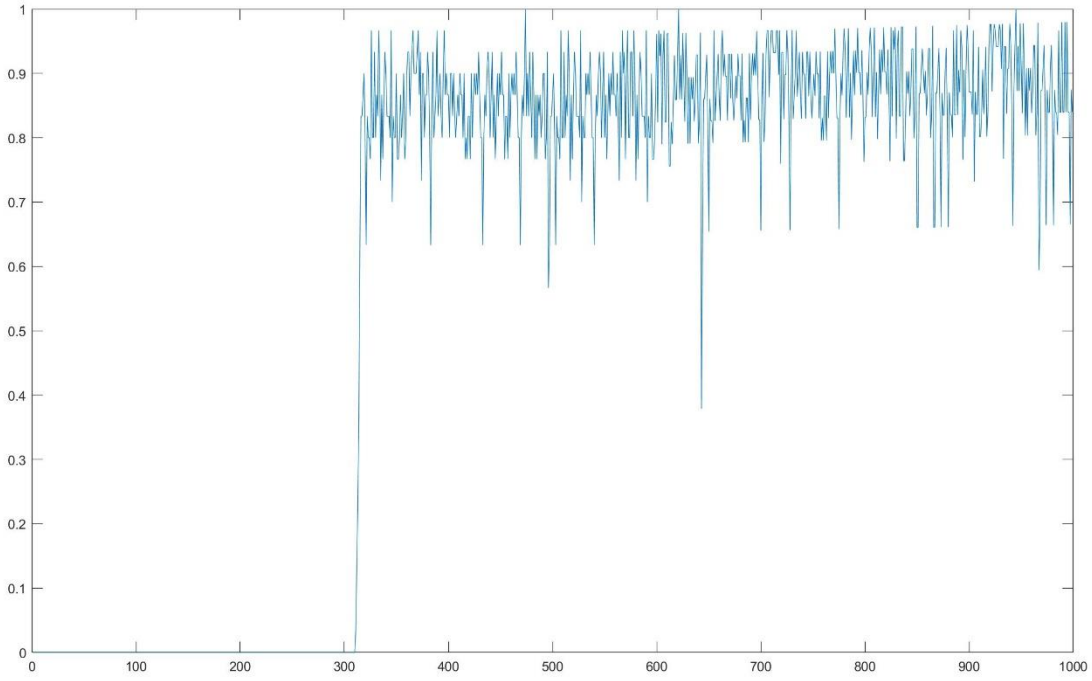
Mutation in genetic programming has multiple ways. In each process, a random number close to 1 might multiply a branch of tree or a number close to 0 add to a branch. Branch can also be replaced by a random number. Besides, operators used in this assignment, which are "+", "-", "*", "/", "sin", "cos", are able to exchange.

Different branches are able to exchange as well, which is the Crossover process. In this assignment, branches that are not in the same layer of a tree can still exchange so that it may improve diversity in population.

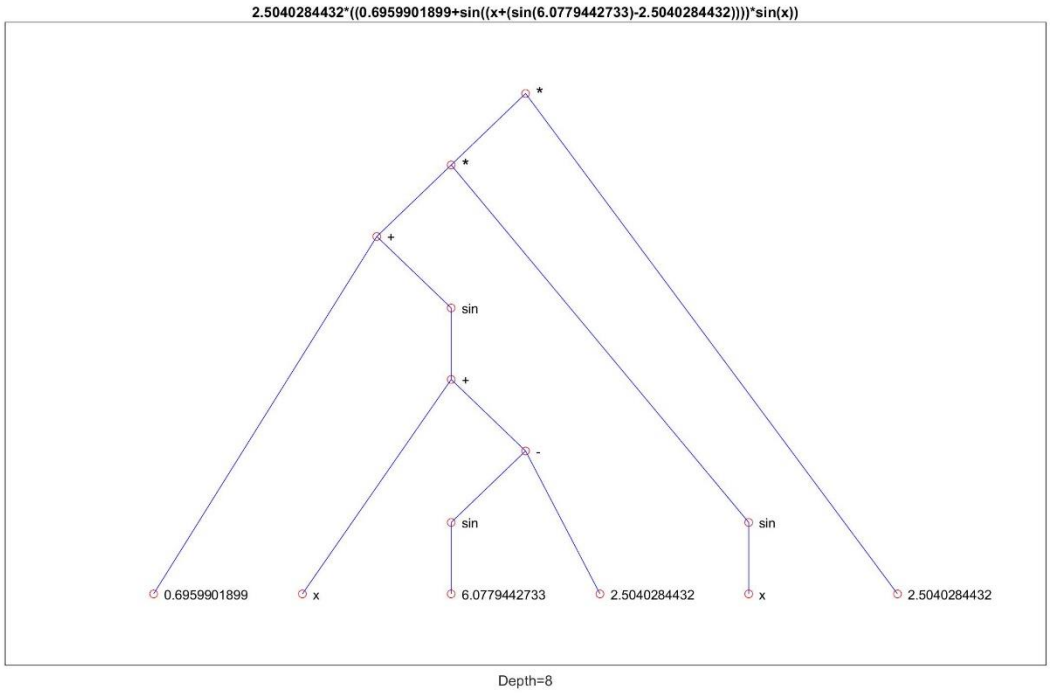
3. Dot plot for GP: High Diversity



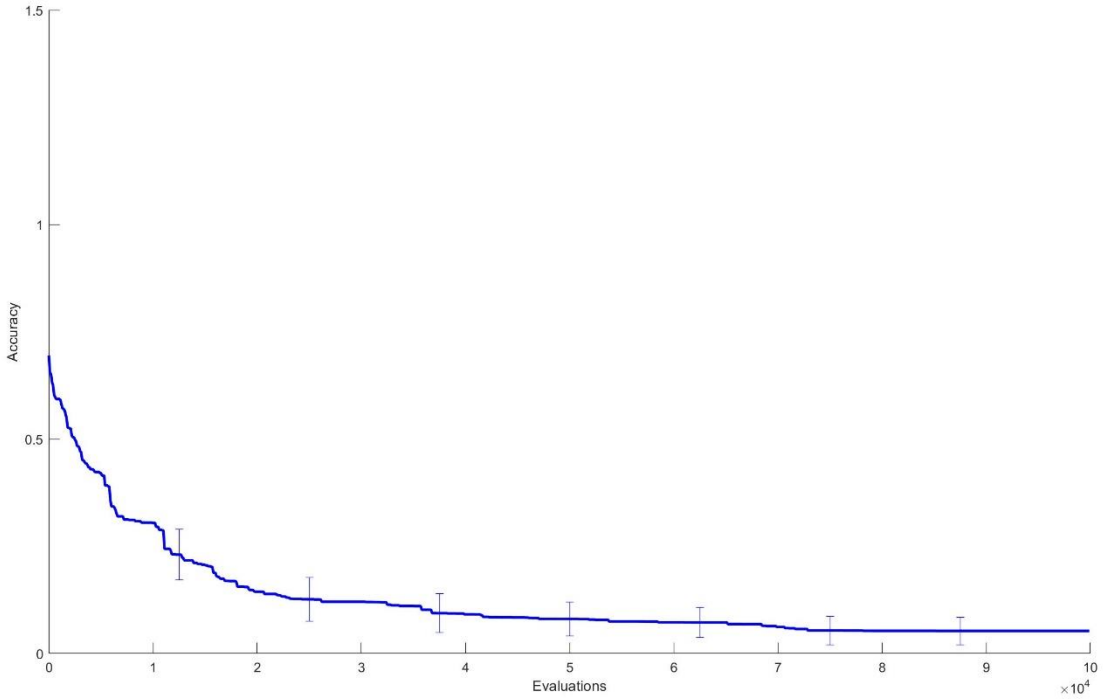
4. Convergence Plot for GP: High Diversity



5. Tree of Best Solution

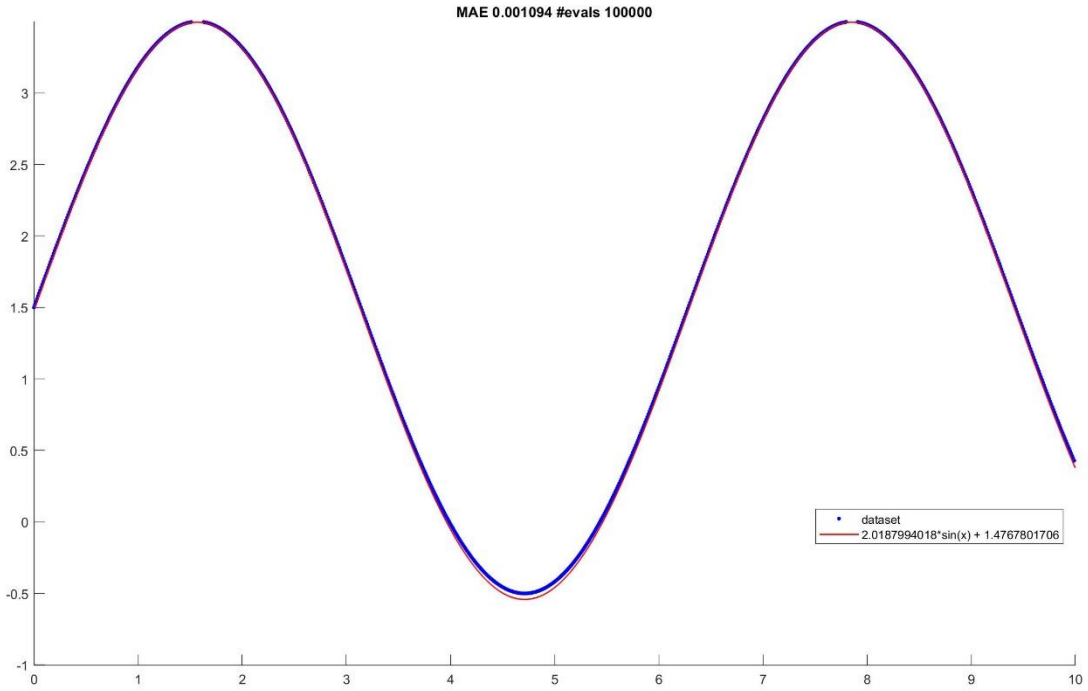


6. Accuracy vs Number of Evaluations



7. Simpler problem tested for debugging

Using Function $f(x) = 2 \times \sin(x) + 1.5$



8. Analysis

From performance plots, the Random Search's best fitness is not better than other methods. After 100000 generations, the best fitness is still higher than 0.5. Hill Climb method is better than random search for this problem. From performance plot, the learning curve of this method has smaller fitness. However, the standard deviation of this method is large which means this method is not stable. After serval times of running hill climber method code, the outcome of this method sometimes quite low while most time the result is not ideal.

GP with low diversity can find a better solution compared with former two methods. This GP method uses Rolette Selection, which means the better an individual's fitness, the higher probability this individual is chosen. In this method, variation is also applied. By using mutation method and crossover method, more outstanding individual has higher probability to come out in population.

GP which uses Truncation Selection is able to find a function whose mean average error is around 0.02, sometimes it can be 0.01 or even lower. Compared with GP using Rolette method it has better performance probably due to Hierarchical Fair Competition. This method is able to improve the diversity of population. After specific generation, the diversity of population decreases, i.e. most individuals are similar. However, Hierarchical Fair Competition method adds some new individuals in population, and compares them with the individuals who has similar fitness so that these new genes will not be eliminated in selection process.

Random Search

```
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <random>
#include <ctime>
#include <iomanip>
#include <string>
#include <cmath>
#include <cstdio>

using namespace std;

#define POINT_NUM 1000
#define GENERATION 100000

void Read();
void Generate();
string To_string(string gene[],int x);
double Operate();

static string operator_dic[8] = { "+","-","*","/","sin","cos","x","a" };
static string calculator[4] = { "+","-","*","/" };
static string tri[2] = { "sin","cos" };
static string cons[2] = { "x","a" };

double xlist[POINT_NUM] = { 0 };
double ylist[POINT_NUM] = { 0 };
double BestYlist[POINT_NUM] = { 0 };
double fitnesslist[GENERATION] = { 0 };
double constant[100] = { 0 };

static string tree[256];

int main()
{
	double fitness = 10000.0;
	double bestfitness = fitness;
	srand(int(time(0)));

	Read();
	for (int loop = 0; loop < GENERATION; loop++)
	{
		for (int i = 0; i < 100; i++)
		{
			constant[i] = 0.0;
		}
		for (int i = 0; i < 256; i++)
		{
			tree[i] = (string)"0";
		}
		Generate();



		for (int i = 0; i < sizeof(tree); i++)
		{
			if (i < (256/2 - 1) && tree[i] == (string)"/" && tree[2 * i + 1] == (string)"0")
			{
				loop--;
				continue;
			}
			if (i < (256 / 2 - 1) && tree[i] == "*" && (tree[2 * i] == "0" || tree[2 * i + 1] ==
"0"))
			{
				loop--;
				continue;
			}
		}

		fitness = Operate();
		int j = 0;
		for (int i = 255; i > 0; i--)
		{
			if (tree[i] == (string)"a")
			{
				char buffer[20];
```

```
                                sprintf_s(buffer, "%.10f", constant[j]);
                                string str = buffer;
                                tree[i] = str;
                                j++;
                        }
                }

                string func = To_string(tree, 1);



                if (bestfitness >= fitness)
                {
                        bestfitness = fitness;
                        cout << "--------------------" << endl;
                        cout << loop << endl;
                        cout << func << endl;
                        cout << fitness << endl;
                }
                fitnesslist[loop] = bestfitness;
        }
        ofstream outputfitness("fitness3.txt");  //import fitness into a txt file

        for (int i = 0; i < GENERATION; i++)
                outputfitness << setprecision(9) << fitnesslist[i] << endl;

        outputfitness.close();
}

void Read()
{
        int i = 0;
        ifstream input("data.txt");

        double a, b;
        while (input >> a >> b) {
                xlist[i] = a;
                ylist[i] = b;
                i++;
        }
        input.close();
}

void Generate()
{
        bool flag = false;
        tree[1] = operator_dic[rand() % 6];
        for (int i = 1; i < 7; i++)
        {
                for (int j = (int)pow(2, i); j < (int)pow(2, i+1); j++)
                {
                        for (int k = 0; k < sizeof(calculator); k++)
                        {
                                if (tree[int(j / 2.0)] == calculator[k])
                                {
                                        tree[j] = operator_dic[rand() % 8];
                                        flag = true;
                                        break;
                                }
                        }

                        if (tree[int(j / 2)] == tri[0] || tree[int(j / 2)] == tri[1])
                        {
                                if (j % 2 == 0)
                                        tree[j] = operator_dic[rand() % 8];
                        }
                }
        }

        for (int i = (int)pow(2, 7); i < (int)pow(2, 8); i++)
        {
                if (tree[int(i / 2)] != (string)"0" && tree[int(i / 2)] != (string)"x" && tree[int(i / 2)] !=
(string)"a")
                {
                        if (tree[int(i / 2)] == (string)"cos" || tree[int(i / 2)] == (string)"sin")
                        {
                                if (i % 2 == 0)
                                {
                                        tree[i] = operator_dic[6 + rand() % 2];
```

```
                        }
                }
                else
                {
                        tree[i] = operator_dic[6 + rand() % 2];
                }
        }
    }
}


string To_string(string gene[], int x)
{
    string function = "\0";
    if ((2 * x >= 255) || (gene[2 * x] == (string)"0"))
        function += gene[x];
    else if (gene[x] == tri[0] || gene[x] == tri[1])
        function = function + gene[x] + (string)"(" + To_string(gene, 2 * x) + (string)")";
    else
        function = function + (string)"(" + To_string(gene, 2 * x) + gene[x] + To_string(gene, 2 * x + 1)
+ (string)")";
    return function;
}

double Operate()
{
    int j = 0;
    for (int i = 0; i < 256; i++)
    {
        if (tree[i] == (string)"a")
        {
            constant[j] = (double)rand() / (double)RAND_MAX * 20.0 - 10.0;
            j++;
        }
    }

    double error = 0.0;
    double fitness[256];
    for (int k = 0; k < POINT_NUM; k++)
    {
        int j = 0;
        for (int i = 0; i < 256; i++)
        {
            fitness[i] = 0.0;
        }

        for (int i = 255; i > 0; i--)
        {
            if (tree[i] == (string)"a")
            {
                fitness[i] = constant[j];
                j++;
            }

            else if (tree[i] == (string)"x")
                fitness[i] = xlist[k];

            else if (tree[i] == (string)"+")
                fitness[i] = fitness[2 * i] + fitness[2 * i + 1];

            else if (tree[i] == (string)"-")
                fitness[i] = fitness[2 * i] - fitness[2 * i + 1];

            else if (tree[i] == (string)"*")
                fitness[i] = fitness[2 * i] * fitness[2 * i + 1];

            else if (tree[i] == (string)"/")
                fitness[i] = fitness[2 * i] / fitness[2 * i + 1];

            else if (tree[i] == (string)"sin")
                fitness[i] = sin(fitness[2 * i]);

            else if (tree[i] == (string)"cos")
                fitness[i] = cos(fitness[2 * i]);
        }
        error += fabs(fitness[1] - ylist[k]);
    }
    return error;
}
```

Hill Climber

```
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <random>
#include <ctime>
#include <iomanip>
#include <string>
#include <cmath>
#include <cstdio>

using namespace std;

#define POINT_NUM 1000
#define GENERATION 100000

void Read();
void Generate();
string To_string(string gene[], int x);
void Clear(string gene[], int x);
double Operate(string gene[]);
void mutation();

static string operator_dic[8] = { "+","-","*","/","sin","cos","x","a" };
static string calculator[4] = { "+","-","*","/" };
static string tri[2] = { "sin","cos" };
static string cons[2] = { "x","a" };

static double xlist[POINT_NUM] = { 0 };
static double ylist[POINT_NUM] = { 0 };
//static double xtest[POINT_NUM] = { 0 };
//static double ytest[POINT_NUM] = { 0 };
static double BestYlist[POINT_NUM] = { 0 };
static double fitnesslist[GENERATION] = { 0 };

static string tree[256];
static string test[256];

int main()
```

```
{
	double fitness = 10000.0;
	double bestfitness = fitness;
	srand(int(time(0)));

	Read();

	bool valid;
	do
	{
		valid = true;
		for (int i = 0; i < 256; i++)
			tree[i] = "0";
		Generate();
		for (int i = 0; i < sizeof(tree); i++)
		{
			if (i < (256 / 2 - 1) && tree[i] == (string)"/" && tree[2 * i + 1] == (string)"0")
			{
				valid = false;
				break;
			}
			if (i < (256 / 2 - 1) && tree[i] == "*" && (tree[2 * i] == "0" || tree[2 * i + 1] ==
"0"))
			{
				valid = false;
				break;
			}
		}
	} while (!valid);
	for (int loop = 0; loop < GENERATION; loop++)
	{

		mutation();
		double f1 = Operate(tree);
		double f2 = Operate(test);
		if (f1 > f2)
		{
			for (int i = 0; i < 256; i++)
				tree[i] = test[i];
		}
		fitness = Operate(tree);
```

```
                string func = To_string(tree, 1);


                if (bestfitness > fitness)
                {
                        bestfitness = fitness;
                        cout << "--------------------" << endl;
                        cout << loop << endl;
                        cout << func << endl;
                        cout << fitness << endl;
                }
                fitnesslist[loop] = bestfitness;
        }
        ofstream outputfitness("HillClimber5.txt");  //import fitness into a txt file


        for (int i = 0; i < GENERATION; i++)
                outputfitness << setprecision(9) << fitnesslist[i] << endl;


        outputfitness.close();
}


void Read()
{
        int i = 0;
        ifstream input("data.txt");
        double x[1000] = { 0 };
        double y[1000] = { 0 };
        double a, b;
        while (input >> a >> b)
        {
                x[i] = a;
                y[i] = b;
                i++;
        }
        input.close();


        for (int i = 0; i < POINT_NUM; i++)
        {
                xlist[i] = x[i];
                ylist[i] = y[i];
```

```
        }


        //      for (int i = 1; i < POINT_NUM; i += 2)
        //      {
        //              xtest[i] = x[i];
        //              ytest[i] = y[i];
        //      }
}
void Generate()
{
        bool flag = false;
        tree[1] = operator_dic[rand() % 6];
        for (int i = 1; i < 7; i++)
        {
                for (int j = (int)pow(2, i); j < (int)pow(2, i + 1); j++)
                {
                        for (int k = 0; k < 4; k++)
                        {
                                if (tree[int(j / 2.0)] == calculator[k])
                                {
                                        tree[j] = operator_dic[rand() % 8];
                                        flag = true;
                                        break;
                                }
                        }

                        if (tree[int(j / 2)] == tri[0] || tree[int(j / 2)] == tri[1])
                        {
                                if (j % 2 == 0)
                                        tree[j] = operator_dic[rand() % 8];
                        }
                }
        }

        for (int i = (int)pow(2, 7); i < (int)pow(2, 8); i++)
        {
                if (tree[int(i / 2)] != (string)"0" && tree[int(i / 2)] != (string)"x" && tree[int(i / 2)] !=
(string)"a")
                {
                        if (tree[int(i / 2)] == (string)"cos" || tree[int(i / 2)] == (string)"sin")
                        {
```

```
                        if (i % 2 == 0)
                        {
                                tree[i] = operator_dic[6 + rand() % 2];
                        }
                }
                else
                {
                        tree[i] = operator_dic[6 + rand() % 2];
                }
        }
    }
    for (int i = 0; i < 256; i++)
    {
        if (tree[i] == "a")
        {
                char buffer[20];
                sprintf_s(buffer, "%.10f", (double)rand() / (double)RAND_MAX * 20.0 - 10.0);
                string str = buffer;
                tree[i] = str;
        }
    }
    //      cout << "Generate Done" << endl;
}
string To_string(string gene[], int x)
{
    string function = "\0";
    if ((2 * x > 256) || (gene[2 * x] == (string)"0"))
        function += gene[x];
    else if (gene[x] == tri[0] || gene[x] == tri[1])
        function = function + gene[x] + (string)"(" + To_string(gene, 2 * x) + (string)")";
    else
        function = function + (string)"(" + To_string(gene, 2 * x) + gene[x] + To_string(gene, 2 * x + 1)
+ (string)")";
    return function;
}


double Operate(string gene[])
{
    double error = 0.0;
    double fitness[256];
    for (int k = 0; k < POINT_NUM; k++)
```

```
{
        int j = 0;
        for (int i = 0; i < 256; i++)
        {
                fitness[i] = 0.0;
        }

        for (int i = 255; i > 0; i--)
        {

                if (gene[i] == "x")
                        fitness[i] = xlist[k];

                else if (gene[i] == "+" && i < 158)
                        fitness[i] = fitness[2 * i] + fitness[2 * i + 1];

                else if (gene[i] == "-" && i < 158)
                        fitness[i] = fitness[2 * i] - fitness[2 * i + 1];

                else if (gene[i] == "*" && i < 158)
                        fitness[i] = fitness[2 * i] * fitness[2 * i + 1];

                else if (gene[i] == "/" && i < 158)
                        fitness[i] = fitness[2 * i] / fitness[2 * i + 1];

                else if (gene[i] == "sin" && i < 158)
                        fitness[i] = sin(fitness[2 * i]);

                else if (gene[i] == "cos" && i < 158)
                        fitness[i] = cos(fitness[2 * i]);

                else if (gene[i] == "0")
                        continue;
                else
                {
                        fitness[i] = atof(gene[i].c_str());
                }

        }
        error += fabs(fitness[1] - ylist[k]);
```

```
        }
        //      cout << "Operate Done" << endl;
        return error;
}

void mutation()
{

        int node = rand() % 256;

        for (int i = 0; i < 256; i++)
                test[i] = tree[i];

        while (true)
        {
                if (test[node] == "0")
                {
                        node = rand() % 256;
                        continue;
                }
                else
                {
                        double ran = (double)rand() / (double)RAND_MAX;
                        if (ran < 0.8)
                        {
                                if (test[node] == "x")
                                {
                                        if (node < 128)
                                        {
                                                int r = rand() % 8;
                                                test[node] = operator_dic[r];
                                                if (r < 4)
                                                {
                                                        //                                                              for
(int i = node * 2; i <= node * 2 + 1; i++)
                                                        //                                                              {
                                                        //
        double r1 = (double)rand() / (double)RAND_MAX;
                                                        //
        if (r1 >= 0.5)
                                                        test[node * 2] = "x";
```

```
                                                //
        else
                                                //
        {
                                                char buffer[20];
                                                sprintf_s(buffer, "%.10f", (double)rand() /
(double)RAND_MAX * 20.0 - 10.0);
                                                string str = buffer;
                                                test[node * 2 + 1] = str;
                                                //
        }
                                                //                                                      }


                                        }
                                        else if (r == 4 || r == 5)
                                        {
                                                //                                                      if
(test[node / 2] == "sin" || test[node / 2] == "cos")
                                                //                                                      {
                                                //
        test[node] = "x";
                                                //                                                      }
                                                //                                                      else
                                                //                                                      {
                                                double r1 = (double)rand() / (double)RAND_MAX;
                                                if (r1 >= 0.5)
                                                        test[node * 2] = "x";


                                                else
                                                {
                                                        char buffer[20];
                                                        sprintf_s(buffer, "%.10f", (double)rand() /
(double)RAND_MAX * 20.0 - 10.0);
                                                        string str = buffer;
                                                        test[node * 2] = str;
                                                }
                                                //                                                      }


                                        }
                                        else if (r == 6)
                                                continue;
                                        else if (r == 7)
                                        {
                                                char buffer[20];
```

```
                                                sprintf_s(buffer, "%.10f", (double)rand() /
(double)RAND_MAX * 20.0 - 10.0);

                                                string str = buffer;

                                                test[node] = str;

                                        }


                                        //                                              cout << "Mutate X" <<
endl;

                                }

                        }

                        else if ((test[node] == "+" || test[node] == "-" || test[node] == "*" ||
test[node] == "/" || test[node] == "sin" || test[node] == "cos") && node < 128)

                        {

                                int r2 = rand() % 8;

                                if (r2 < 4 && (test[node] == "sin" || test[node] == "cos"))

                                {

                                        double r3 = (double)rand() / (double)RAND_MAX;

                                        if (r3 >= 0.5)

                                        {

                                                char buffer[20];

                                                sprintf_s(buffer, "%.10f", (double)rand() /
(double)RAND_MAX * 20.0 - 10.0);

                                                string str = buffer;

                                                test[2 * node + 1] = str;

                                        }

                                        else

                                                test[2 * node + 1] = "x";

                                        test[node] = operator_dic[r2];

                                }

                                else if (r2 >= 4 && r2 <= 5 && (test[node] == "+" || test[node] == "-" ||
test[node] == "*" || test[node] == "/"))

                                {

                                        Clear(test, 2 * node + 1);

                                        test[node] = operator_dic[r2];

                                }


                                else if (r2 == 6)

                                {

                                        Clear(test, node);

                                        if (node > 1)

                                        {

                                                test[node] = operator_dic[r2];

                                        }
```

```
                }
                else if (r2 == 7)
                {
                        Clear(test, node);
                        if (node > 1)
                        {
                                char buffer[20];
                                sprintf_s(buffer, "%.10f", (double)rand() /
(double)RAND_MAX * 20.0 - 10.0);
                                string str = buffer;
                                test[node] = str;
                        }
                }


                //                                      cout << "Mutate Operator" << endl;
        }
        else if (test[node] == "0")
                continue;
        else
        {
                double r = (double)rand() / (double)RAND_MAX;
                if (r < 0.5)
                {
                        double c = atof(test[node].c_str());
                        if ((double)rand() / (double)RAND_MAX >= 0.5)
                                c += (double)rand() / (double)RAND_MAX;

                        else
                                c -= (double)rand() / (double)RAND_MAX;

                        if (c > 10.0)
                                c -= 10.0;
                        else if (c < -10.0)
                                c += 10.0;

                        char buffer[20];
                        sprintf_s(buffer, "%.10f", c);
                        string str = buffer;
                        test[node] = str;
                }
```

```
                                        else
                                        {
                                                if (node < 128)
                                                {
                                                        int r = rand() % 8;


                                                        if (r < 4)
                                                        {
                                                                test[node] = operator_dic[r];
                                                                //
        for (int i = node * 2; i <= node * 2 + 1; i++)
                                                                //
        {
                                                                //
                double r1 = (double)rand() / (double)RAND_MAX;
                                                                //
                if (r1 >= 0.5)
                                                                test[2 * node] = "x";


                                                                //
                else
                                                                //
                {
                                                                char buffer[20];
                                                                sprintf_s(buffer, "%.10f", (double)rand() /
(double)RAND_MAX * 20.0 - 10.0);
                                                                string str = buffer;
                                                                test[2 * node + 1] = str;
                                                                //
                }
                                                                //
        }


                                                        }
                                                        else if (r == 4 || r == 5)
                                                        {


                                                                //
        if (test[node / 2] != "sin" && test[node / 2] != "cos")
                                                                //
        {
                                                                test[node] = operator_dic[r];
                                                                double r1 = (double)rand() / (double)RAND_MAX;
                                                                if (r1 >= 0.5)
                                                                        test[node * 2] = "x";
```

```
                                                        else
                                                        {
                                                                char buffer[20];
                                                                sprintf_s(buffer, "%.10f", (double)rand() /
(double)RAND_MAX * 20.0 - 10.0);
                                                                string str = buffer;
                                                                test[node * 2] = str;
                                                        }

                                                        //
        }
                                                }
                                                else if (r == 6)
                                                {
                                                        test[node] = "x";
                                                }
                                                else if (r == 7)
                                                        continue;
                                        }
                                }
                        }
                        else if (ran >= 0.8 && node > 1)
                        {
                                string tank = "\0";
                                Clear(test, node);
                                test[node] = "x";
                        }
                        break;
                }
        }
}
void Clear(string gene[], int x)
{
        gene[x] = "0";
        if (2 * x < 256)
        {
                Clear(gene, 2 * x);
                Clear(gene, 2 * x + 1);
        }
}
```

GP

GPmain.cpp

```
#include "GP.h"
#include <iostream>
#include <fstream>
#include <ctime>
#include <iomanip>
#include <algorithm>
#include <string>
using namespace std;


int main()
{
	Population p;
	static double pathlength[GENERATION] = { 0.0 };
	int i = 0;
	srand(int(time(0)));

	ifstream input("data.txt");

	double a, b;
	static double x[1000] = { 0 }, y[1000] = { 0 };
	while (input >> a >> b) {
		x[i] = a;
		y[i] = b;
		i++;
	}
	input.close();
	int g = 0;
	for (int i = 0; i < 1000; i += 10)
	{
		p.xlist[g] = x[i];
		p.ylist[g] = y[i];
		g++;
	}


	p.Initial_Population(POPULATION_SIZE);

	p.BestIndividual.push_back(p.Population[0]);

	srand(int(time(0)));
	ofstream outputdot("conv.txt");
	for (p.loop = 0; p.loop < GENERATION; p.loop++)
	{

		double fit = p.BestIndividual[0].Fitness;
		p.TotalFit();
		p.Selection();
		outputdot << a << endl;
		p.Crossover();
		p.Mutation();
		p.Population.clear();
		sort(p.NewPopulation.begin(), p.NewPopulation.end());
		for (int s = 0; s < POPULATION_SIZE; s++)
			p.Population.push_back(p.NewPopulation[s]);

		p.CompareFit();
		p.Simplify();

		if (p.BestIndividual[0].Fitness < fit)
		{
			cout << "Generation" << p.loop + 1 << ":" << p.BestIndividual[0].Fitness << endl;
			cout << p.To_string(p.BestIndividual[0].Genome, 1) << endl;
		}
		pathlength[p.loop] = p.BestIndividual[0].Fitness;
		if (p.loop % 10 == 0)
		{
			cout << "Generation" << p.loop + 1 << endl;
		}
		for (int i = 0; i < POPULATION_SIZE; i++)
		{
			for (int j = 0; j < 256; j++)
			{
				if ((p.NewPopulation[i].Genome[j] == "+" || p.NewPopulation[i].Genome[j] == "-" ||
p.NewPopulation[i].Genome[j] == "/"
```

```
                                        || p.NewPopulation[i].Genome[j] == "*" ) && p.NewPopulation[i].Genome[2 *
j] == "0" && p.NewPopulation[i].Genome[2*j+1] == "0")
                                {
                                        if (i == 0)
                                        {
                                                p.NewPopulation[i] = p.NewPopulation[i + 1];
                                        }
                                        else
                                        {
                                                p.NewPopulation[i] = p.NewPopulation[0];

                                        }
                                }
                        }
                }

                int a = 0;
                for (int i = 0; i < POPULATION_SIZE; i++)
                {
                        if (p.NewPopulation[i].Fitness < 0.3)
                        {
                                a++;
                        }
                }

        }

        outputdot.close();
        ofstream outputfitness("fitness.txt");

        for (int i = 0; i < GENERATION; i++)
        {
                outputfitness << setprecision(9) << pathlength[i] << endl;
        }

        outputfitness.close();

        p.NewPopulation.clear();
        p.Population.clear();
        return 0;
}
```

GP.h

```
#define _GP_H_
#include<vector>
#include<cmath>
#include<string>

#define POPULATION_SIZE 100
#define POINT_NUM 100
#define GENERATION 1000
#define MUTATION 0.2
#define CROSSOVER 0.7

using namespace std;

class Individual
{
public:
        friend class Population;

        Individual() :Fitness(0) {}
        Individual(vector<string> genome, double fitness) : Genome(genome), Fitness(fitness) {}
        ~Individual() {}

        vector<string> Genome;

        double Fitness;

        bool operator < (const Individual& I)const
        {
                return Fitness < I.Fitness;
        }
};

class Population
{
public:
        void Initial_Population(int P);
        void Crossover();
        void Selection();
        void Mutation();
        void Adaptive();
        void TotalFit();
        void TotalFit2();
        void CompareFit();
        void Clear(int i, int x);
        void Save(vector<string> gene, int x);
        void Copy(int x, int y, int node1, int node2);
        void Copy2(int x, vector<string> gene, int node1, int node2);
        int IsLong(int Ind, int node);
        string To_string(vector<string> gene, int x);
        double Distance(vector<string> gene1, vector<string> gene2);
        void Simplify();
        //double Similar(vector<string> gene1, vector<string> gene2);

        double Fitness(vector<string> gene);
        double totalfitness = 0.0;
        double totalfitnessInv = 0.0;
        double averagefitness;
        double pastfitness;
        double P = 0.02;
        double xlist[POINT_NUM] = { 0.0 }, ylist[POINT_NUM] = { 0.0 };

        int Generation;
        int loop;
        vector<double> PC;
        vector<double> PM;
        vector<string> Cage;
        vector<Individual> Population;
        vector<Individual> NewPopulation;
        vector<Individual> BestIndividual;
        string operator_dic[8] = { "+","-","*","/","sin","cos","x","a" };
        string calculator[4] = { "+","-","*","/" };
        string tri[2] = { "sin","cos" };
        string cons[2] = { "x","a" };
};
```

GP.cpp

```
#include <iostream>
#include <fstream>
#include <ctime>
#include <cstdio>
#include <random>
#include <vector>
#include <map>
#include <algorithm>
#include <iomanip>
#include <typeinfo>
#include <chrono>
#include <string>
#include "GP.h"


using namespace std;


void Population::Initial_Population(int P)
{
	vector<Individual> Genometemp;
	Genometemp.clear();
	for (int i = 0; i < P; i++)
	{
		int num;
		for (num = 0; num < POPULATION_SIZE; num++)
		{
			Genometemp.push_back(Individual());
			for (int j = 0; j < 256; j++)
				Genometemp[num].Genome.push_back("0");

			Genometemp[num].Genome[1] = operator_dic[rand() % 6];
			for (int i = 1; i < 7; i++)
			{
				for (int j = (int)pow(2, i); j < (int)pow(2, i + 1); j++)
				{
					for (int k = 0; k < 4; k++)
					{
						if (Genometemp[num].Genome[int(j / 2.0)] == calculator[k])
```

```
                                        {
                                                Genometemp[num].Genome[j] = operator_dic[rand() % 8];
                                                break;
                                        }
                                }

                                if (Genometemp[num].Genome[int(j / 2)] == tri[0] ||
Genometemp[num].Genome[int(j / 2)] == tri[1])
                                {
                                        if (j % 2 == 0)
                                                Genometemp[num].Genome[j] = operator_dic[rand() % 8];
                                }
                        }
                }

                for (int i = (int)pow(2, 7); i < (int)pow(2, 8); i++)
                {
                        if (Genometemp[num].Genome[int(i / 2)] != (string)"0" &&
Genometemp[num].Genome[int(i / 2)] != (string)"x" && Genometemp[num].Genome[int(i / 2)] != (string)"a")
                        {
                                if (Genometemp[num].Genome[int(i / 2)] == (string)"cos" ||
Genometemp[num].Genome[int(i / 2)] == (string)"sin")
                                {
                                        if (i % 2 == 0)
                                        {
                                                Genometemp[num].Genome[i] = operator_dic[6 + rand() % 2];
                                        }
                                }
                                else
                                {
                                        Genometemp[num].Genome[i] = operator_dic[6 + rand() % 2];
                                }
                        }
                }

                for (int i = 0; i < 256; i++)
                {
                        if (Genometemp[num].Genome[i] == "a")
                        {
                                char buffer[20];
                                sprintf_s(buffer, "%.10f", (double)rand() / (double)RAND_MAX * 20.0 -
10.0);
```

```
                                string str = buffer;
                                Genometemp[num].Genome[i] = str;
                        }
                }

        }

        for (int i = 0; i < POPULATION_SIZE; ++i)
        {
                Genometemp[i].Fitness = Fitness(Genometemp[i].Genome);
        }
        sort(Genometemp.begin(), Genometemp.end());
        Population.push_back(Genometemp[0]);
        Genometemp.clear();
    }
}


double Population::Fitness(vector<string> gene)
{
    double error = 0.0;
    double fitness[256];
    for (int k = 0; k < POINT_NUM; k++)
    {
        for (int i = 0; i < 256; i++)
            fitness[i] = 0.0;


        for (int i = 255; i > 0; i--)
        {

            if (gene[i] == "x")
                fitness[i] = xlist[k];

            else if (gene[i] == "+" && i < 158)
                fitness[i] = fitness[2 * i] + fitness[2 * i + 1];

            else if (gene[i] == "-" && i < 158)
                fitness[i] = fitness[2 * i] - fitness[2 * i + 1];
```

```
                        else if (gene[i] == "*" && i < 158)
                                fitness[i] = fitness[2 * i] * fitness[2 * i + 1];

                        else if (gene[i] == "/" && i < 158)
                                fitness[i] = fitness[2 * i] / fitness[2 * i + 1];

                        else if (gene[i] == "sin" && i < 158)
                                fitness[i] = sin(fitness[2 * i]);

                        else if (gene[i] == "cos" && i < 158)
                                fitness[i] = cos(fitness[2 * i]);

                        else if (gene[i] == "0")
                                continue;
                        else
                                fitness[i] = atof(gene[i].c_str());
                }
                error += fabs(fitness[1] - ylist[k])/POINT_NUM;
        }
        return error;
}

#if 1
//Truncation Selection
void Population::Selection()
{
        sort(Population.begin(), Population.end());
        NewPopulation.clear();
    NewPopulation.push_back(BestIndividual[0]);
        vector<Individual>::iterator it;
        for (it = Population.begin(); it != Population.end();)
        {
                if (it - Population.begin() >= 80)
                {
                        it = Population.erase(it);
                }
                else
                {
                        it++;
                }
```

```
        }
        int a = Population.size();
        Initial_Population(POPULATION_SIZE - a);



        for (int i = 0; i < POPULATION_SIZE; i++)
        {
                if ((int)(i / 10) % 2 == 0)
                {
                        NewPopulation.push_back(Population[i]);
                }
        }

}
#endif

#if 0
void Population::Selection()
{
        double sum;
        double point;
        sort(Population.begin(), Population.end());
        NewPopulation.clear();
//      NewPopulation.push_back(BestIndividual[0]);
        NewPopulation.push_back(Population[0]);

        while (NewPopulation.size() != POPULATION_SIZE / 2)
        {
                sum = 0.0;
                point = ((double)rand() / (double)RAND_MAX) * totalfitnessInv;

                vector<Individual>::iterator it;
                for (it = Population.begin(); it != Population.end(); it++)
                {
                        sum += 1.0 / (*(it)).Fitness;
                        if (sum >= point)
                        {
                                NewPopulation.push_back(*it);
                                break;
                        }
```

```
                }
        }
}
#endif

void Population::TotalFit()
{
        totalfitnessInv = 0.0;
        for (int i = 0; i < POPULATION_SIZE; i++)
        {
                totalfitnessInv += 1.0 / Population[i].Fitness;
        }
}

void Population::TotalFit2()
{
        totalfitness = 0.0;
        for (unsigned int i = 0; i < NewPopulation.size(); i++)
        {
                totalfitness += NewPopulation[i].Fitness;
        }
        averagefitness = totalfitness / (double)NewPopulation.size();
}

#if 0
//PMX
void Population::Crossover()
{
        int a = 0, b = 0, x, y;
        bool same = true;
        double Pc_;
        int i = 0;
        sort(NewPopulation.begin(), NewPopulation.end());
        TotalFit2();
        Adaptive();

        do
        {
                NewPopulation.push_back(NewPopulation[i]);
                i++;
```

```
} while (NewPopulation.size() != POPULATION_SIZE);


for (int i = 0; i < POPULATION_SIZE / 5; i++)
{
        do
        {
                while (true)
                {
                        x = rand() % (POPULATION_SIZE / 2);
                        Pc_ = (double)rand() / (double)RAND_MAX;
                        if (Pc_ <= PC[x])
                                break;
                }
                while (true)
                {
                        y = rand() % (POPULATION_SIZE / 2);
                        Pc_ = (double)rand() / (double)RAND_MAX;
                        if (Pc_ <= PC[y])
                                break;
                }
        } while (x == y);

        int node1,node2;
        do
        {
                do
                {
                        node1 = rand() % 255 + 1;
                } while (NewPopulation[x].Genome[node1] == "0");

                do
                {
                        node2 = rand() % 255 + 1;
                } while (NewPopulation[y].Genome[node2] == "0");


        } while ( (int)(log(node1)/log(2)) != (int)(log(node2) / log(2)));


        Cage.clear();
```

```
                for (int i = 0; i < 256; i++)
                        Cage.push_back("0");

                Save(NewPopulation[x].Genome, node1);

                Clear(NewPopulation[x].Genome, node1);

                Copy(NewPopulation[x].Genome, NewPopulation[y].Genome, node1, node2);

                Clear(NewPopulation[y].Genome, node2);

                Copy(NewPopulation[y].Genome, Cage, node2, node1);

                NewPopulation[x].Fitness = Fitness(NewPopulation[x].Genome);
                NewPopulation[y].Fitness = Fitness(NewPopulation[y].Genome);
        }

}
#endif

#if 1
void Population::Crossover()
{
        int a = 0, b = 0, x, y;
        double Pc_;
        int i = 0;
        sort(NewPopulation.begin(), NewPopulation.end());

        do
        {
                NewPopulation.push_back(NewPopulation[i]);
                i++;
        } while (NewPopulation.size() != POPULATION_SIZE);

        vector<int> cross;
        cross.clear();
        for (int i = 0; i < POPULATION_SIZE / 2; i++)
                cross.push_back(i);
        unsigned seed = (unsigned)chrono::system_clock::now().time_since_epoch().count();
        shuffle(cross.begin(), cross.end(), default_random_engine(seed));
```

```
	for (int i = 0; i < POPULATION_SIZE / 2; i += 2)
	{
		Pc_ = (double)rand() / (double)RAND_MAX;
		x = cross[i];
		y = cross[i + 1];
		if(Pc_ < CROSSOVER)
		{

			int node1, node2;
			do
			{
				do
				{
					node1 = rand() % 255 + 1;
				} while (NewPopulation[x].Genome[node1] == "0");

				do
				{
					node2 = rand() % 255 + 1;
				} while (NewPopulation[y].Genome[node2] == "0");

			} while (IsLong(x, node1) + (int)(log(node2) / log(2)) >= 7 || IsLong(y, node2) +
(int)(log(node1) / log(2)) >= 7);

			Cage.clear();
			for (int i = 0; i < 256; i++)
				Cage.push_back("0");

			Save(NewPopulation[x].Genome, node1);

			Clear(x, node1);


			Copy(x, y, node1, node2);
			Clear(y, node2);

			Copy2(y, Cage, node2, node1);

			NewPopulation[x].Fitness = Fitness(NewPopulation[x].Genome);
```

```
                        NewPopulation[y].Fitness = Fitness(NewPopulation[y].Genome);



                        //Deterministic Crowding
                        if (Distance(NewPopulation[x].Genome, NewPopulation[x + POPULATION_SIZE / 2].Genome) + \
                                Distance(NewPopulation[y].Genome, NewPopulation[y + POPULATION_SIZE / 2].Genome) <
\
                                Distance(NewPopulation[y].Genome, NewPopulation[x + POPULATION_SIZE / 2].Genome) +
\
                                Distance(NewPopulation[x].Genome, NewPopulation[y + POPULATION_SIZE / 2].Genome))
                        {
                                if (NewPopulation[x].Fitness < NewPopulation[x + POPULATION_SIZE / 2].Fitness)
                                {
                                        NewPopulation[x + POPULATION_SIZE / 2] = NewPopulation[x];
                                }
                                if (NewPopulation[y].Fitness < NewPopulation[y + POPULATION_SIZE / 2].Fitness)
                                {
                                        NewPopulation[y + POPULATION_SIZE / 2] = NewPopulation[y];
                                }
                        }
                        else
                        {
                                if (NewPopulation[x].Fitness < NewPopulation[y + POPULATION_SIZE / 2].Fitness)
                                {
                                        NewPopulation[y + POPULATION_SIZE / 2] = NewPopulation[x];
                                }
                                if (NewPopulation[y].Fitness < NewPopulation[x + POPULATION_SIZE / 2].Fitness)
                                {
                                        NewPopulation[x + POPULATION_SIZE / 2] = NewPopulation[y];
                                }
                        }



                }
        }
}
#endif

#if 0
void Population::Mutation()
{
```

```
        double Pm_ = 0.0;

        sort(NewPopulation.begin(), NewPopulation.end());
        TotalFit2();
        Adaptive();
        int ind;
        for (ind = 0; ind < POPULATION_SIZE; ind++)
        {

                Pm_ = (double)rand() / (double)RAND_MAX;

                if (Pm_ <= PM[ind])
                {
                        while (true)
                        {
                                int node = rand() % 256;
                                if (NewPopulation[ind].Genome[node] == "0")
                                        continue;
                                else
                                {
                                        double ran = (double)rand() / (double)RAND_MAX;
                                        if (ran < 0.9)
                                        {
                                                if (NewPopulation[ind].Genome[node] == "x")
                                                {
                                                        if (node < 128 && node > 0)
                                                        {
                                                                int r = rand() % 8;
                                                                NewPopulation[ind].Genome[node] = operator_dic[r];
                                                                if (r < 4)
                                                                {
                                                                        for (int i = node * 2; i <= node * 2 + 1;
i++)
                                                                        {
                                                                                double r1 = (double)rand() /
(double)RAND_MAX;
                                                                                if (r1 >= 0.5)
                                                                                        NewPopulation[ind].Genome[i]
= "x";

                                                                                else
                                                                                {
```

```
                                                                char buffer[20];
                                                                sprintf_s(buffer, "%.10f",
(double)rand() / (double)RAND_MAX * 20.0 - 10.0);
                                                                string str = buffer;
                                                                NewPopulation[ind].Genome[i]
= str;
                                                        }
                                                }
                                        }
                                        else if (r == 4 || r == 5)
                                        {
                                                double r1 = (double)rand() /
(double)RAND_MAX;
                                                if (r1 >= 0.5)
                                                        NewPopulation[ind].Genome[node * 2] =
"x";

                                                else
                                                {
                                                        char buffer[20];
                                                        sprintf_s(buffer, "%.10f",
(double)rand() / (double)RAND_MAX * 20.0 - 10.0);
                                                        string str = buffer;
                                                        NewPopulation[ind].Genome[node * 2] =
str;
                                                }
                                        }
                                        else if (r == 6)
                                                NewPopulation[ind].Genome[node] == "x";
                                        else if (r == 7)
                                        {
                                                char buffer[20];
                                                sprintf_s(buffer, "%.10f", (double)rand() /
(double)RAND_MAX * 20.0 - 10.0);
                                                string str = buffer;
                                                NewPopulation[ind].Genome[node] = str;
                                        }
                                }
                        }
                        else if ((NewPopulation[ind].Genome[node] == "+" \
                                || NewPopulation[ind].Genome[node] == "-" \
                                || NewPopulation[ind].Genome[node] == "*" \
                                || NewPopulation[ind].Genome[node] == "/" \
                                || NewPopulation[ind].Genome[node] == "sin" \
```

```
                                                || NewPopulation[ind].Genome[node] == "cos") && node < 128)
                                        {
                                                int r2 = rand() % 8;
                                                if (r2 < 4 && (NewPopulation[ind].Genome[node] == "sin" ||
NewPopulation[ind].Genome[node] == "cos"))
                                                {
                                                        double r3 = (double)rand() / (double)RAND_MAX;
                                                        if (r3 >= 0.5)
                                                        {
                                                                char buffer[20];
                                                                sprintf_s(buffer, "%.10f", (double)rand() /
(double)RAND_MAX * 20.0 - 10.0);
                                                                string str = buffer;
                                                                NewPopulation[ind].Genome[2 * node + 1] =
str;
                                                        }
                                                        else
                                                                NewPopulation[ind].Genome[2 * node + 1] =
"x";
                                                        NewPopulation[ind].Genome[node] = operator_dic[r2];
                                                }
                                                else if (r2 >= 4 && r2 <= 5 &&
(NewPopulation[ind].Genome[node] == "+" || \
                                                        NewPopulation[ind].Genome[node] == "-" || \
                                                        NewPopulation[ind].Genome[node] == "*" || \
                                                        NewPopulation[ind].Genome[node] == "/"))
                                                {
                                                        Clear(ind, 2 * node + 1);
                                                        NewPopulation[ind].Genome[node] = operator_dic[r2];
                                                }

                                                else if (r2 == 6)
                                                {
                                                        if (node > 1 && node < 256)
                                                        {
                                                                Clear(ind, node);
                                                                NewPopulation[ind].Genome[node] =
operator_dic[r2];
                                                        }
                                                }
                                                else if (r2 == 7)
                                                {
                                                        if (node > 1 && node < 256)
```

```
                {
                        Clear(ind, node);
                        char buffer[20];
                        sprintf_s(buffer, "%.10f", (double)rand() /
(double)RAND_MAX * 20.0 - 10.0);
                        string str = buffer;
                        NewPopulation[ind].Genome[node] = str;
                }
        }
}
else if (NewPopulation[ind].Genome[node] == "0")
        continue;
else
{
        double r = (double)rand() / (double)RAND_MAX;
        if (r < 0.5)
        {
                double c =
atof(NewPopulation[ind].Genome[node].c_str());
                if ((double)rand() / (double)RAND_MAX >= 0.5)
                        c += (double)rand() / (double)RAND_MAX;

                else
                        c -= (double)rand() / (double)RAND_MAX;

                if (c > 10.0)
                        c -= 10.0;
                else if (c < -10.0)
                        c += 10.0;

                char buffer[20];
                sprintf_s(buffer, "%.10f", c);
                string str = buffer;
                NewPopulation[ind].Genome[node] = str;
        }
        else
        {
                int r = rand() % 8;
                if (node < 128)
                {
                        if (r < 4)
```

```
operator_dic[r];

+ 1; i++)

(double)RAND_MAX;

        NewPopulation[ind].Genome[i] = "x";

"%.10f", (double)rand() / (double)RAND_MAX * 20.0 - 10.0);

        NewPopulation[ind].Genome[i] = str;

operator_dic[r];

(double)RAND_MAX;

        NewPopulation[ind].Genome[node * 2] = "x";

(double)rand() / (double)RAND_MAX * 20.0 - 10.0);

        NewPopulation[ind].Genome[node * 2] = str;

{
        NewPopulation[ind].Genome[node] =

        for (int i = node * 2; i <= node * 2

        {
                double r1 = (double)rand() /

                if (r1 >= 0.5)

                else
                {
                        char buffer[20];
                        sprintf_s(buffer,

                        string str = buffer;

                }
        }
}
else if (r == 4 || r == 5)
{

        NewPopulation[ind].Genome[node] =

        double r1 = (double)rand() /

        if (r1 >= 0.5)

        else
        {
                char buffer[20];
                sprintf_s(buffer, "%.10f",

                string str = buffer;

        }
}
else if (r == 6)
```

```
                                                                {
                                                                        NewPopulation[ind].Genome[node] =
"x";
                                                                }
                                                                else if (r == 7)
                                                                        continue;
                                                        }
                                                }
                                        }
                                }
                                else if (ran >= 0.9 && node > 1)
                                {
                                        string tank = "\0";
                                        Clear(ind, node);
                                        NewPopulation[ind].Genome[node] = "x";
                                }
                                break;
                        }
                }
        }
        NewPopulation[ind].Fitness = Fitness(NewPopulation[ind].Genome);
    }
}
#endif

#if 1
void Population::Mutation()
{
    double Pm_ = 0.0;

    sort(NewPopulation.begin(), NewPopulation.end());
//  TotalFit2();
//  Adaptive();
    int ind;
    for (ind = 0; ind < POPULATION_SIZE; ind++)
    {
        Pm_ = (double)rand() / (double)RAND_MAX;

//          if (Pm_ <= PM[ind])
        if(Pm_ <= MUTATION)
        {
```

```
                        while (true)
                        {
                                int node = rand() % 256;
                                if (NewPopulation[ind].Genome[node] == "0")
                                        continue;
                                else
                                {
                                        double x = (double)rand() / (double)RAND_MAX;
                                        if (x < 0.5)
                                        {
                                                if ((NewPopulation[ind].Genome[node] == "x" || \
                                                        NewPopulation[ind].Genome[node] == "+" || \
                                                        NewPopulation[ind].Genome[node] == "-" || \
                                                        NewPopulation[ind].Genome[node] == "*" || \
                                                        NewPopulation[ind].Genome[node] == "/" || \
                                                        NewPopulation[ind].Genome[node] == "sin" || \
                                                        NewPopulation[ind].Genome[node] == "cos") && node < 128)
                                                {
                                                        int pin = rand() % 3;
                                                        int len = IsLong(ind, node);
                                                        if ((pin == 0) && (log(node) / log(2) + len < 8))
                                                        {
                                                                Cage.clear();
                                                                for (int i = 0; i < 256; i++)
                                                                        Cage.push_back("0");
                                                                Save(Cage, node);
                                                                Clear(ind, node);
                                                                Copy2(ind, Cage, 2 * node, node);
                                                                NewPopulation[ind].Genome[node] = "+";
                                                                char buffer[20];
                                                                sprintf_s(buffer, "%.10f", ((double)rand() /
(double)RAND_MAX - 0.5) * 0.1);
                                                                string str = buffer;
                                                                NewPopulation[ind].Genome[2 * node + 1] = str;
                                                        }
                                                        else if ((pin == 1) && (log(node) / log(2) + len < 8))
                                                        {
                                                                Cage.clear();
                                                                for (int i = 0; i < 256; i++)
                                                                        Cage.push_back("0");
                                                                Save(Cage, node);
```

```
                Clear(ind, node);
                Copy2(ind, Cage, 2 * node, node);
                NewPopulation[ind].Genome[node] = "*";
                char buffer[20];
                sprintf_s(buffer, "%.10f", ((double)rand() /
(double)RAND_MAX - 0.5) * 0.1 + 1);
                string str = buffer;
                NewPopulation[ind].Genome[2 * node + 1] = str;
            }
            else if (pin == 2)
            {
                Clear(ind, node);
                char buffer[20];
                sprintf_s(buffer, "%.10f", (double)rand() /
(double)RAND_MAX * 20.0 - 10.0);
                string str = buffer;
                NewPopulation[ind].Genome[node] = str;
            }
        }
        else
        {
            double x = (double)rand() / (double)RAND_MAX;
            if (x < 0.6)
            {
                double c =
atof(NewPopulation[ind].Genome[node].c_str());
                if ((double)rand() / (double)RAND_MAX >= 0.5)
                    c += (double)rand() / (double)RAND_MAX *
0.5;

                else
                    c -= (double)rand() / (double)RAND_MAX *
0.5;

                if (c > 10.0)
                    c -= 10.0;
                else if (c < -10.0)
                    c += 10.0;

                char buffer[20];
                sprintf_s(buffer, "%.10f", c);
                string str = buffer;
                NewPopulation[ind].Genome[node] = str;
```

```
                                        }
                                        else
                                        {
                                                NewPopulation[ind].Genome[node] = "x";
                                        }


                                }
                        }
                        else
                        {
                                if ((NewPopulation[ind].Genome[node] == "+" \
                                        || NewPopulation[ind].Genome[node] == "-" \
                                        || NewPopulation[ind].Genome[node] == "*" \
                                        || NewPopulation[ind].Genome[node] == "/" \
                                        || NewPopulation[ind].Genome[node] == "sin" \
                                        || NewPopulation[ind].Genome[node] == "cos") && node < 128)
                                {
                                        int r2 = rand() % 8;
                                        if (r2 < 4 && (NewPopulation[ind].Genome[node] == "sin" ||
NewPopulation[ind].Genome[node] == "cos"))
                                        {
                                                NewPopulation[ind].Genome[2 * node + 1] = "x";
                                                NewPopulation[ind].Genome[node] = operator_dic[r2];
                                        }
                                        else if (r2 >= 4 && r2 <= 5 &&
(NewPopulation[ind].Genome[node] == "+" || NewPopulation[ind].Genome[node] == "-" ||
NewPopulation[ind].Genome[node] == "*" || NewPopulation[ind].Genome[node] == "/")
                                                && ((NewPopulation[ind].Genome[(int)(node / 2)] !=
"sin" && NewPopulation[ind].Genome[(int)(node / 2)] != "cos") || \
                                                 (NewPopulation[ind].Genome[(int)(node / 4)] !=
"sin" && NewPopulation[ind].Genome[(int)(node / 4)] != "cos")))
                                        {
                                                Clear(ind, 2 * node + 1);
                                                NewPopulation[ind].Genome[node] = operator_dic[r2];
                                        }


                                        else if (r2 == 6)
                                        {
                                                if (node > 1 && node < 256)
                                                {
                                                        Clear(ind, node);
                                                        NewPopulation[ind].Genome[node] =
operator_dic[r2];
```

```
                                                        }
                                                }
                                                /*
                                                else if (r2 == 7)
                                                {
                                                        continue;

                                                        if (node > 1 && node < 256)
                                                        {
                                                                Clear(ind, node);
                                                                char buffer[20];
                                                                sprintf_s(buffer, "%.10f", (double)rand() /
(double)RAND_MAX * 20.0 - 10.0);
                                                                string str = buffer;
                                                                NewPopulation[ind].Genome[node] = str;
                                                        }

                                                }
                                                */
                                        }
                                }
                                break;
                        }
                }
                NewPopulation[ind].Fitness = Fitness(NewPopulation[ind].Genome);
        }
}
#endif



void Population::CompareFit()
{
        sort(NewPopulation.begin(), NewPopulation.end());
        if (BestIndividual[0].Fitness > NewPopulation[0].Fitness)
                BestIndividual[0] = NewPopulation[0];
}



void Population::Adaptive()
{
```

```
        PC.clear();
        PM.clear();
        int x = NewPopulation.size();
        for (int i = 0; i < x; i++)
        {
                if (NewPopulation[i].Fitness > averagefitness)
                {
                        PC.push_back(0.9);
                        PM.push_back(0.1);
                }
                else
                {
                        PC.push_back(0.9 - (0.9 - 0.6) * (averagefitness - NewPopulation[i].Fitness) /
(averagefitness - NewPopulation[0].Fitness));
                        PM.push_back(0.1 - (0.1 - 0.001) * (averagefitness - NewPopulation[i].Fitness) /
(averagefitness - NewPopulation[0].Fitness));
                }
        }
}


void Population::Clear(int i, int x)
{
        NewPopulation[i].Genome[x] = "0";
        if (2 * x < 256)
        {
                Clear(i, 2 * x);
                Clear(i, 2 * x + 1);
        }
}


void Population::Save(vector<string> gene, int x)
{
        Cage[x] = gene[x];
        if (2 * x < 256)
        {
                Save(gene, 2 * x);
                Save(gene, 2 * x + 1);
        }
}


void Population::Copy(int x, int y, int node1, int node2)
```

```
{
        NewPopulation[x].Genome[node1] = NewPopulation[y].Genome[node2];
        if (2 * max(node1,node2) < 256)
        {
                Copy(x, y, 2 * node1, 2 * node2);
                Copy(x, y, 2 * node1 + 1, 2 * node2 + 1);
        }
}

void Population::Copy2(int x, vector<string> gene, int node1, int node2)
{
        NewPopulation[x].Genome[node1] = gene[node2];
        if (2 * max(node1,node2) < 256)
        {
                Copy2(x, gene, 2 * node1, 2 * node2);
                Copy2(x, gene, 2 * node1 + 1, 2 * node2 + 1);
        }
}

string Population::To_string(vector<string> gene, int x)
{
        string function = "\0";
        if ((2 * x > 256) || (gene[2 * x] == (string)"0"))
                function += gene[x];
        else if (gene[x] == tri[0] || gene[x] == tri[1])
                function = function + gene[x] + (string)"(" + To_string(gene, 2 * x) + (string)")";
        else
                function = function + (string)"(" + To_string(gene, 2 * x) + gene[x] + To_string(gene, 2 * x + 1)
+ (string)")";
        return function;
}

int Population::IsLong(int Ind, int node)
{
        int i = 1;
        int len = 1;
        while (node * pow(2, i) < 256)
        {
                int j = node * (int)pow(2, i);
                int k = 1;
                while (k < (int)pow(2, i))
```

```
		{
			if (NewPopulation[Ind].Genome[j] != "0")
			{
				len++;
				break;
			}
			j++;
			k++;
		}
		i++;
	}
	return len;
}

double Population::Distance(vector<string> gene1, vector<string> gene2)
{
	double error = 0.0;
	double fitness[256] = { 0.0 };
	double fitness2[256] = { 0.0 };
	for (int k = 0; k < POINT_NUM; k += 10)
	{
		for (int i = 0; i < 256; i++)
			fitness[i] = 0.0;

		for (int i = 255; i > 0; i--)
		{

			if (gene1[i] == "x")
				fitness[i] = xlist[k];

			else if (gene1[i] == "+" && i < 158)
				fitness[i] = fitness[2 * i] + fitness[2 * i + 1];

			else if (gene1[i] == "-" && i < 158)
				fitness[i] = fitness[2 * i] - fitness[2 * i + 1];

			else if (gene1[i] == "*" && i < 158)
				fitness[i] = fitness[2 * i] * fitness[2 * i + 1];

			else if (gene1[i] == "/" && i < 158)
```

```
                fitness[i] = fitness[2 * i] / fitness[2 * i + 1];

        else if (gene1[i] == "sin" && i < 158)
                fitness[i] = sin(fitness[2 * i]);

        else if (gene1[i] == "cos" && i < 158)
                fitness[i] = cos(fitness[2 * i]);

        else if (gene1[i] == "0")
                continue;
        else
                fitness[i] = atof(gene1[i].c_str());
}

for (int i = 0; i < 256; i++)
        fitness2[i] = 0.0;



for (int i = 255; i > 0; i--)
{

        if (gene2[i] == "x")
                fitness2[i] = xlist[k];

        else if (gene2[i] == "+" && i < 158)
                fitness2[i] = fitness2[2 * i] + fitness2[2 * i + 1];

        else if (gene2[i] == "-" && i < 158)
                fitness2[i] = fitness2[2 * i] - fitness2[2 * i + 1];

        else if (gene2[i] == "*" && i < 158)
                fitness2[i] = fitness2[2 * i] * fitness2[2 * i + 1];

        else if (gene2[i] == "/" && i < 158)
                fitness2[i] = fitness2[2 * i] / fitness2[2 * i + 1];

        else if (gene2[i] == "sin" && i < 158)
                fitness2[i] = sin(fitness2[2 * i]);

        else if (gene2[i] == "cos" && i < 158)
```

```
                        fitness2[i] = cos(fitness2[2 * i]);


                else if (gene2[i] == "0")
                        continue;
                else
                        fitness2[i] = atof(gene2[i].c_str());
        }


        error += fabs(fitness[1] - fitness2[1]) / 10;
    }
    return error;
}


void Population::Simplify()
{
    if (BestIndividual[0].Fitness < 0.05)
    {
        for (int i = 0; i < 256; i++)
        {
            if (BestIndividual[0].Genome[i] == "+" && BestIndividual[0].Genome[2 * i].size() > 8 &&
BestIndividual[0].Genome[2 * i + 1].size() > 8)
            {
                double a = atof(BestIndividual[0].Genome[2 * i].c_str());
                double b = atof(BestIndividual[0].Genome[2 * i + 1].c_str());
                BestIndividual[0].Genome[2 * i] = "0";
                BestIndividual[0].Genome[2 * i + 1] = "0";
                char buffer[20];
                sprintf_s(buffer, "%.10f", a + b);
                string str = buffer;
                BestIndividual[0].Genome[i] = str;
            }
            else if (BestIndividual[0].Genome[i] == "-" && BestIndividual[0].Genome[2 * i].size() > 8
&& BestIndividual[0].Genome[2 * i + 1].size() > 8)
            {
                double a = atof(BestIndividual[0].Genome[2 * i].c_str());
                double b = atof(BestIndividual[0].Genome[2 * i + 1].c_str());
                BestIndividual[0].Genome[2 * i] = "0";
                BestIndividual[0].Genome[2 * i + 1] = "0";
                char buffer[20];
                sprintf_s(buffer, "%.10f", a - b);
                string str = buffer;
```

```
                        BestIndividual[0].Genome[i] = str;

                }

                else if (BestIndividual[0].Genome[i] == "*" && BestIndividual[0].Genome[2 * i].size() > 8
&& BestIndividual[0].Genome[2 * i + 1].size() > 8)

                {

                        double a = atof(BestIndividual[0].Genome[2 * i].c_str());

                        double b = atof(BestIndividual[0].Genome[2 * i + 1].c_str());

                        BestIndividual[0].Genome[2 * i] = "0";

                        BestIndividual[0].Genome[2 * i + 1] = "0";

                        char buffer[20];

                        sprintf_s(buffer, "%.10f", a * b);

                        string str = buffer;

                        BestIndividual[0].Genome[i] = str;

                }

                else if (BestIndividual[0].Genome[i] == "/" && BestIndividual[0].Genome[2 * i].size() > 8
&& BestIndividual[0].Genome[2 * i + 1].size() > 8)

                {

                        double a = atof(BestIndividual[0].Genome[2 * i].c_str());

                        double b = atof(BestIndividual[0].Genome[2 * i + 1].c_str());

                        BestIndividual[0].Genome[2 * i] = "0";

                        BestIndividual[0].Genome[2 * i + 1] = "0";

                        char buffer[20];

                        sprintf_s(buffer, "%.10f", a/b);

                        string str = buffer;

                        BestIndividual[0].Genome[i] = str;

                }

                else if (BestIndividual[0].Genome[i] == "sin" && BestIndividual[0].Genome[2 * i].size() >
8)

                {

                        double a = atof(BestIndividual[0].Genome[2 * i].c_str());

                        BestIndividual[0].Genome[2 * i] = "0";

                        char buffer[20];

                        sprintf_s(buffer, "%.10f", sin(a));

                        string str = buffer;

                        BestIndividual[0].Genome[i] = str;

                }

                else if (BestIndividual[0].Genome[i] == "cos" && BestIndividual[0].Genome[2 * i].size() >
8)

                {

                        double a = atof(BestIndividual[0].Genome[2 * i].c_str());

                        BestIndividual[0].Genome[2 * i] = "0";

                        char buffer[20];
```

```
                        sprintf_s(buffer, "%.10f", cos(a));

                        string str = buffer;

                        BestIndividual[0].Genome[i] = str;

                    }

                    else if (BestIndividual[0].Genome[i] == "-" && (BestIndividual[0].Genome[2 * i] ==
BestIndividual[0].Genome[2 * i + 1]))

                    {

                        BestIndividual[0].Genome[2 * i] = "0";

                        BestIndividual[0].Genome[2 * i + 1] = "0";

                        BestIndividual[0].Genome[i] = "0.000000000";

                    }

                    else if (BestIndividual[0].Genome[i] == "/" && (BestIndividual[0].Genome[2 * i] ==
BestIndividual[0].Genome[2 * i + 1]))

                    {

                        BestIndividual[0].Genome[2 * i] = "0";

                        BestIndividual[0].Genome[2 * i + 1] = "0";

                        BestIndividual[0].Genome[i] = "1.000000000";

                    }

                }

            }

        }
```