MECS 4510
Evolutionary Computation and Design Automation

Xinsheng Gu | UNI: xg2381
Instructor: Hod Lipson
Date submitted: 10/26/2021
Grace hours used: 0 hour
Grace hours remaining: 96 hours

1 Results Summary

Table 1: Results Summary

Method	Evaluations	Best Fitness (MAE)
Random Search	50000	1.9073
Hill Climber	50000	1.5823
Genetic Programming (Pm = 0.1)	5000	0.57042
Genetic Programming (Pm = 0.3)	5000	0.37492
Genetic Programming (Pm = 0.3)	50000	0.21857
Genetic Programming (Pm = 0.5)	5000	0.47748

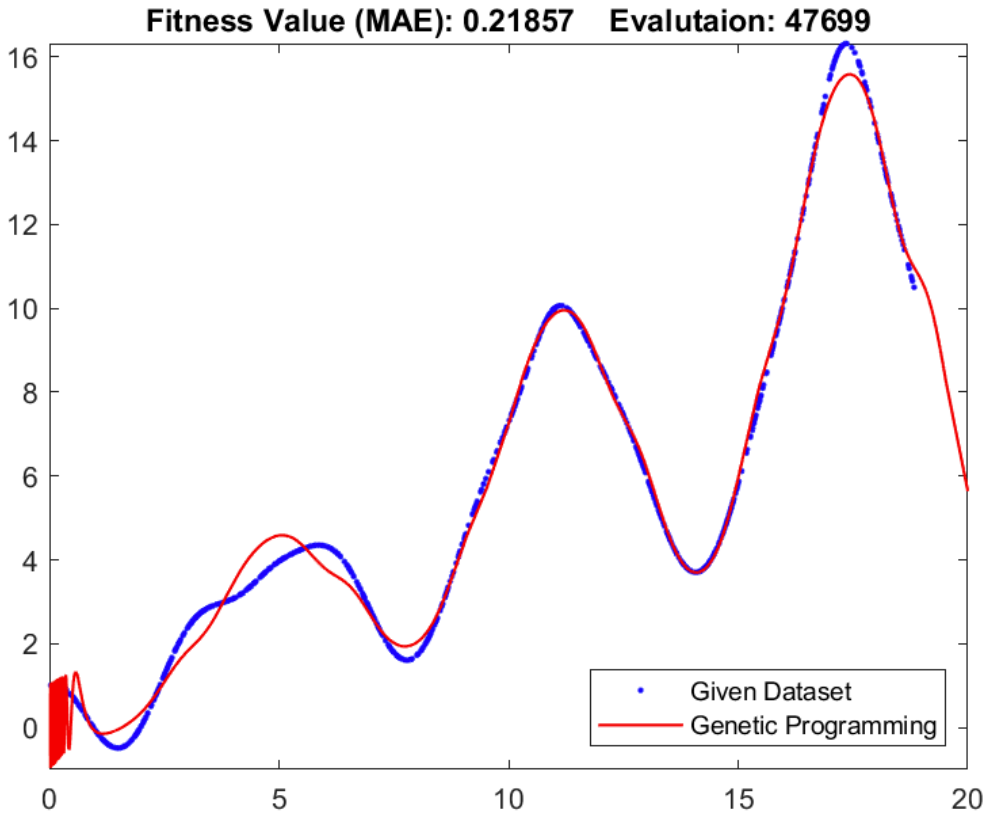

Fig 1 The best results function plot compared with the given dataset. It is done by Genetic Programming with a population size of 100 and a mutation rate of 0.3, under 50000 times generations.

The simplified function searched by GP is:

$$f(x) = 0.58 * x - 1.0 * sin(x + 1.4/x\text{^}2 - 0.012) * (0.29 * x + 1.4/(x + 1.3)) - (3.0 * sin(4.1 * sin(x)))/(x - 23.0)$$

2 Methods

2.1 Representation: I used a heap, which is a numerical array in MATLAB, to represent constants, independent variable x, operators, as well as the relationship between these elements. To be elaborate, since the symbolic regression is essentially a data structure as binary trees, in my heap, for a node whose index is n, its left and right children's index are 2*n and (2*n + 1), respectively. For a binary tree with a D-layer depth, it has (2^D - 1) elements.

2.2 Random Search: In order to create a D-layer perfect binary tree, the random heap is divided into two parts. The first part is its last layer consisting of independent variable x and constants, created randomly. The second part is the layers from the first to the $(D-1)^{th}$ layer, where all its elements are operators. Though 'sin' and 'cos' are single operators while '+', '-', '*', '/' are binary operators, in order to be align with the perfect binary tree, I made 'sin' and 'cos' become binary operators here, which means sin(a, b) = a*sin(b), and cos(a, b) = a*cos(b), so that I can simplify the data structure. Having done with the random create process, the created heap is calculated by another function where its expression is compared with the given dataset and then I obtain the fitness (MAE) of this heap. After that another heap is created and then its fitness is calculated. By comparing fitness value with the former one, the heap can stay if it is better than the former one, while the heap may also be discarded if the former one is better.

2.3 Hill Climber: I used random creation mentioned above to create a population with NP individuals. Then, I did the mutation method with a set mutation rate (Pm). That means for a population, there is a percentage of Pm individual(s) will be chose to slightly change its operators and constants or independent variables. After doing mutation, the fitness of each individual as well as the unmutated individual is calculated one by one and sorted according to their fitness value and keep the best half individuals and discard the others. Loop the process above until a set maximum generation is met.

2.4 EA Variation operators: Crossover and mutation method are carried out as variation operators. For the crossover method, individuals in the population are randomly matched into pairs of parents. Then, one crossover point is randomly selected from the node index from two to the last one. Since the crossover should between the node and all of its children, I used recursion function to search all the children for a branch. Therefore, the crossover is done between the same positions of parent heaps. The mutation method is the same as the one in Hill Climber.

2.5 EA Selection method: The selection method is done based on the fitness of individuals in the population. After calculated all the fitness for every individual, the individuals are sorted according to their fitness. The top certain percentage of individuals are selected into the next generation while the others are discarded.

2.6 Analysis of performance: Compared with the performance of Random Search, Hill Climber and Genetic Programming, I found GP is better than RS and HC, which means GP can search solution faster and more likely to hit a better solution. Although RS and HC are not as good as GP, they still have some edges. For instance, RS is a global search which is less likely to hit a plateau; while HC can search an optimum quickly during the first hundred of iterations.

For GP, the bigger times of iteration, the more likely it is to find better solution, which also takes much longer time to calculate. A low mutation rate may let GP stuck in a local optimum, while a high mutation rate would make GP so random that it is far away from evolution algorithm.

3 Performance Plots

3.1 Learning Curve (RS, HC, GP)

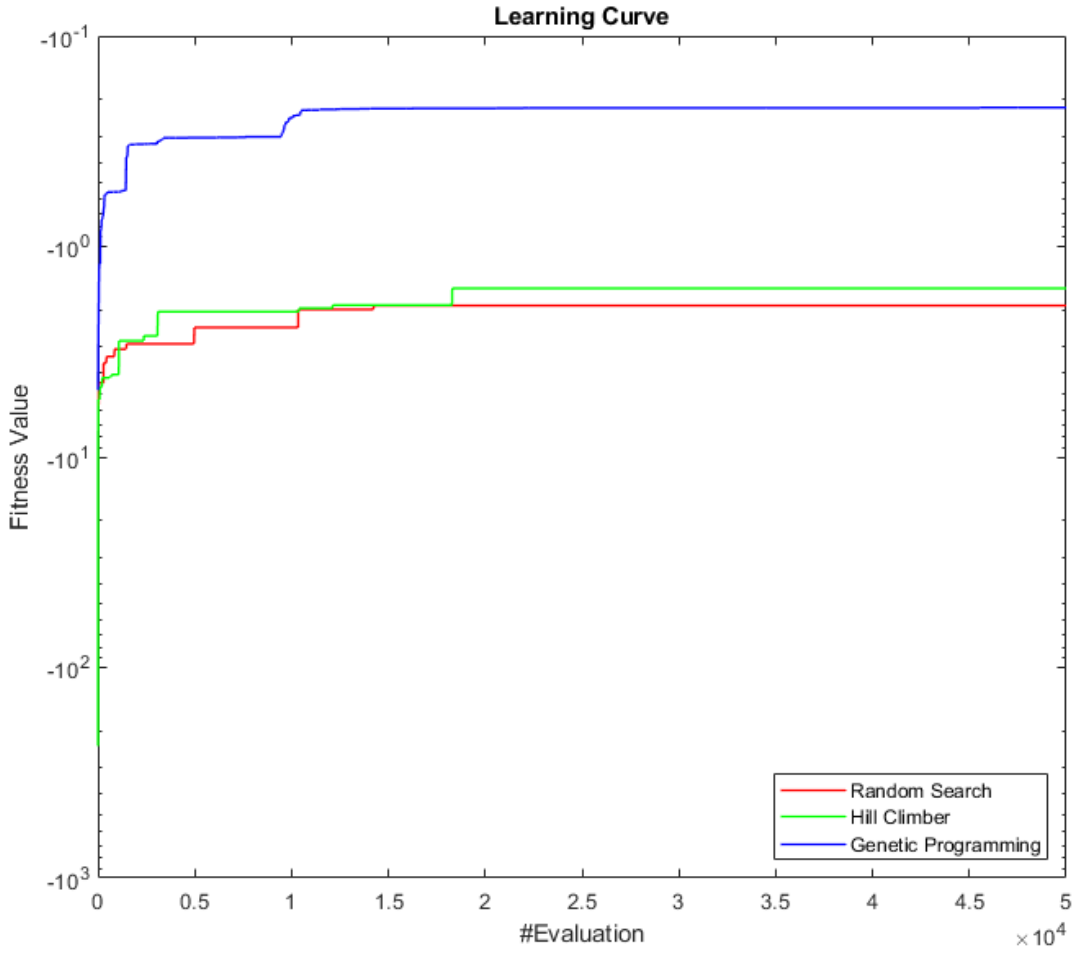

Fig 2 Learning Curve for different methods (RS, HC, GP), where the fitness value stands for mean absolute error

3.2 Learning Curve of GP with different mutation rate

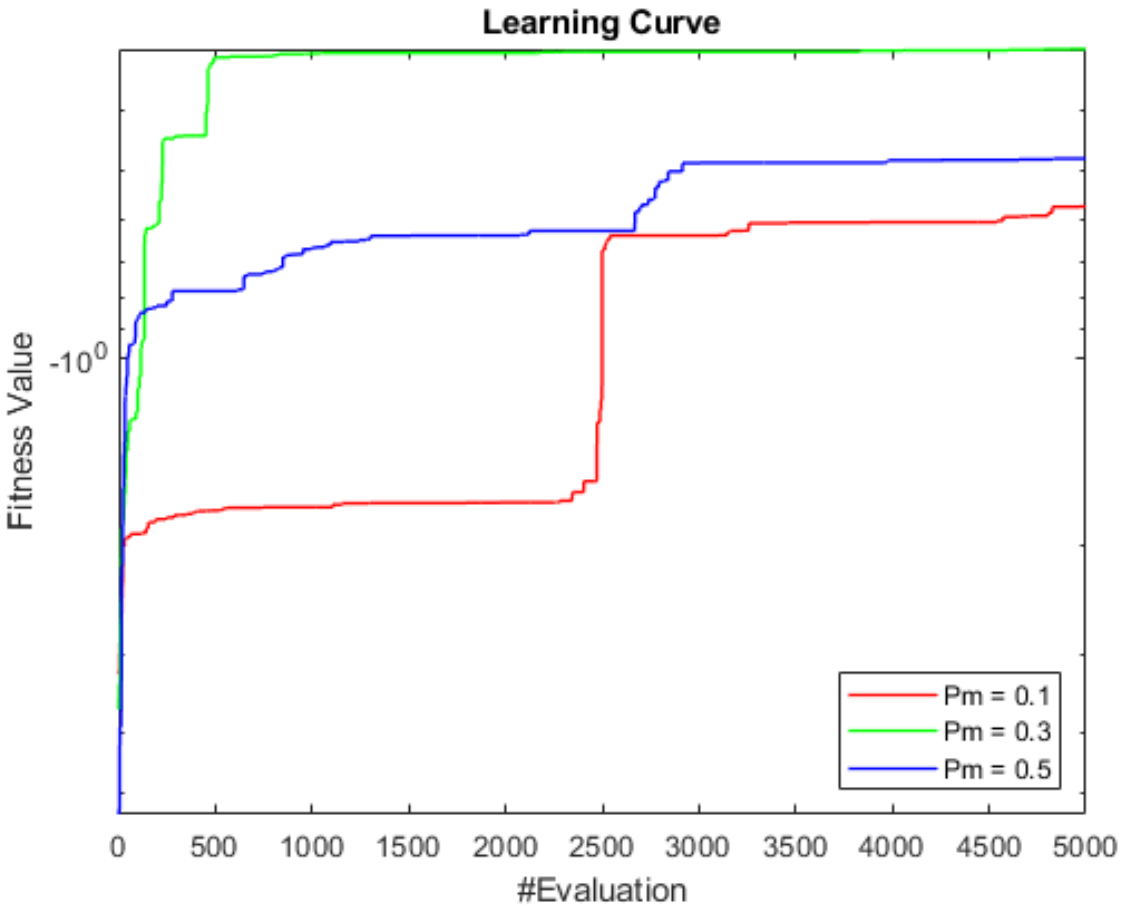

Fig 3 Learning Curve of GP with different mutation rate, where we can find that neither a high mutation rate nor a low mutation rate would achieve a good result

3.3 Dot Plot

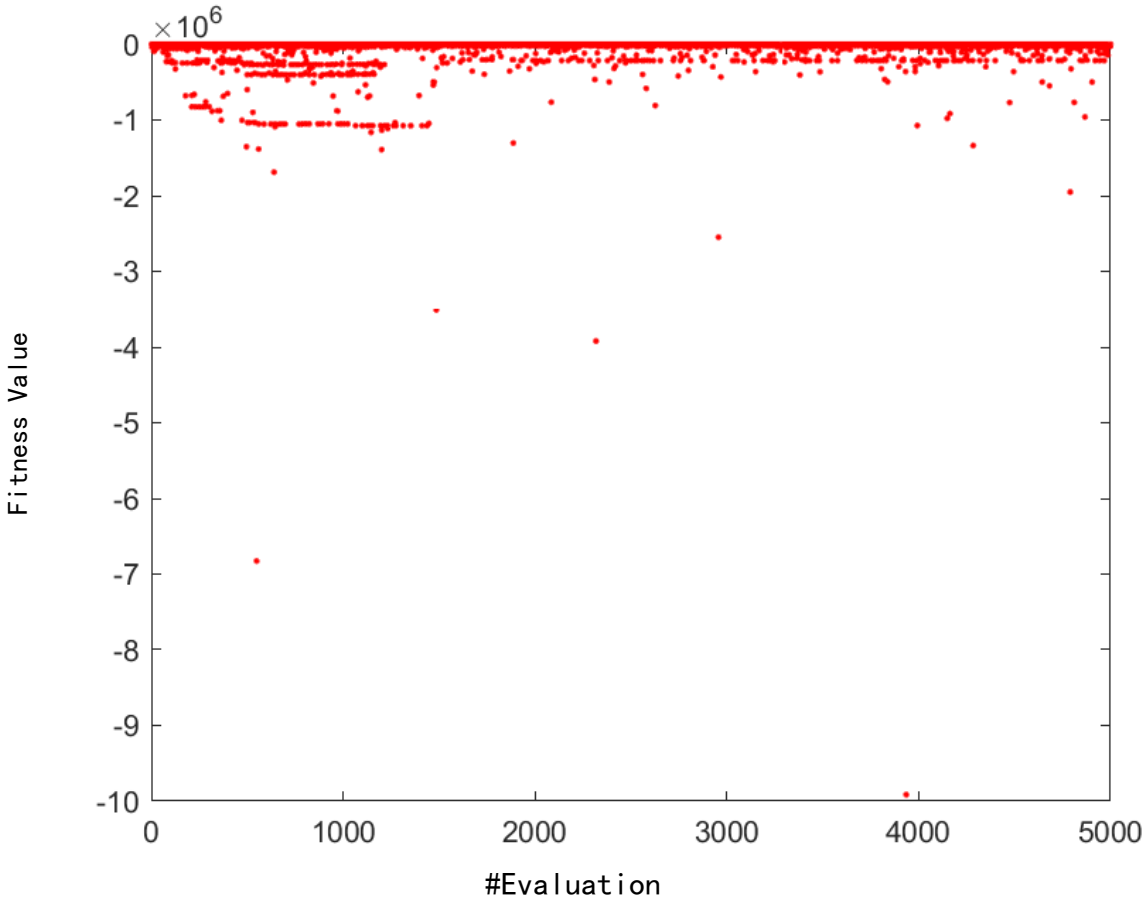

Fig 4 Dot Plot for Genetic Programming (Pm = 0.3). We can see that with the times of evaluation,

3.4 Diversity Plot

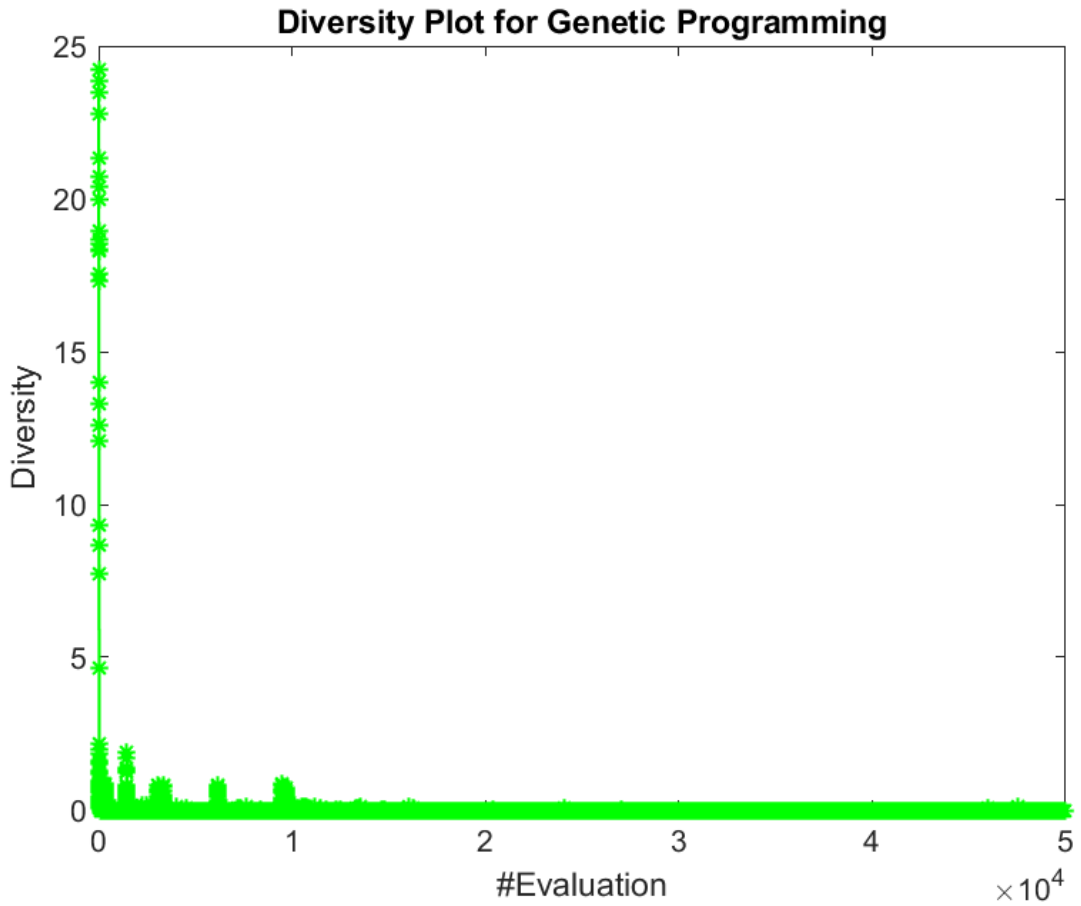

Fig 5 Diversity Plot for Genetic Programming. We can see that the diversity dropped rapidly during the first thousand evaluations, and stay in a relatively low diversity plateau.

3.5 Convergence Plot

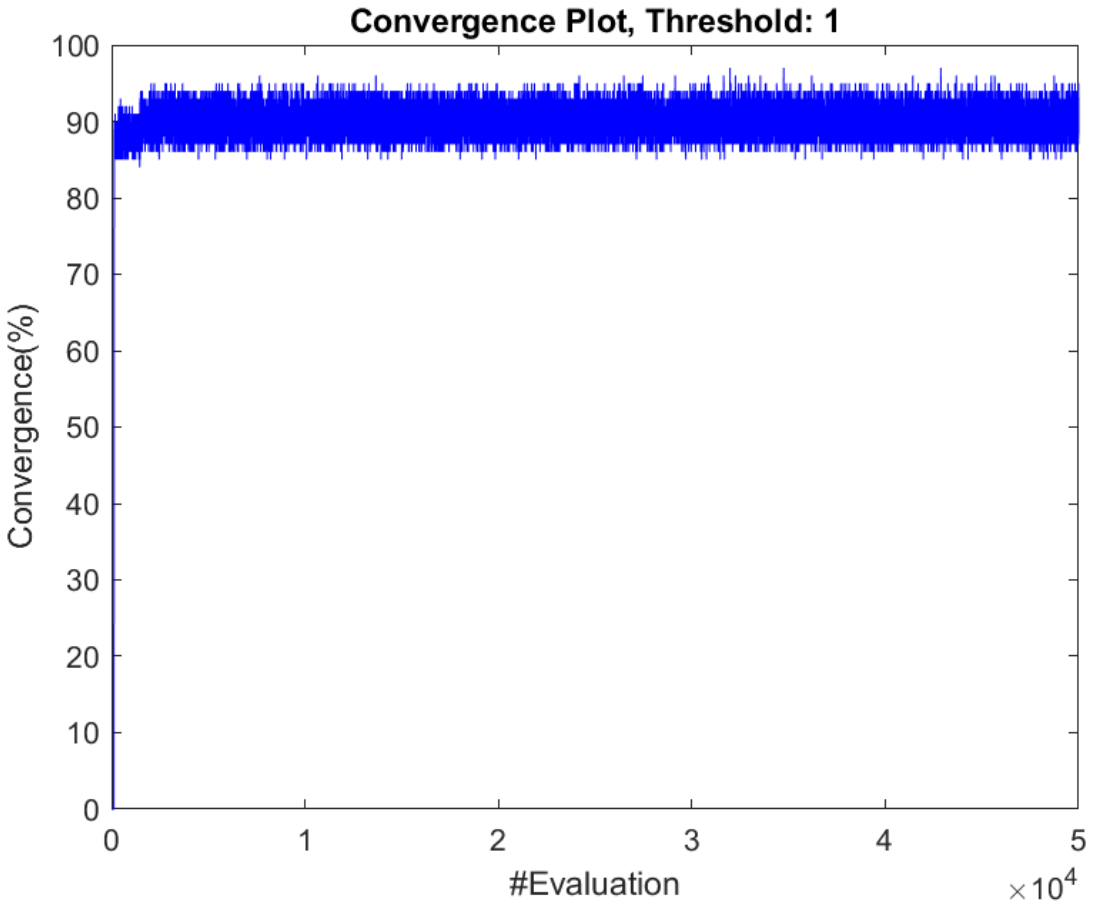

Fig 6 Convergence plot for Genetic Algorithm. The threshold is set at 1.

3.6 Simpler Problem Tested

In order to help myself do test and debug, I created a new dataset consisting 50 data points comes from a simple function expression:

```
y = x + sin(x).
```

After a relatively small times of evaluation, the GP can achieve a pretty good result so that I am confirmed that my code is ready to solve the given dataset.

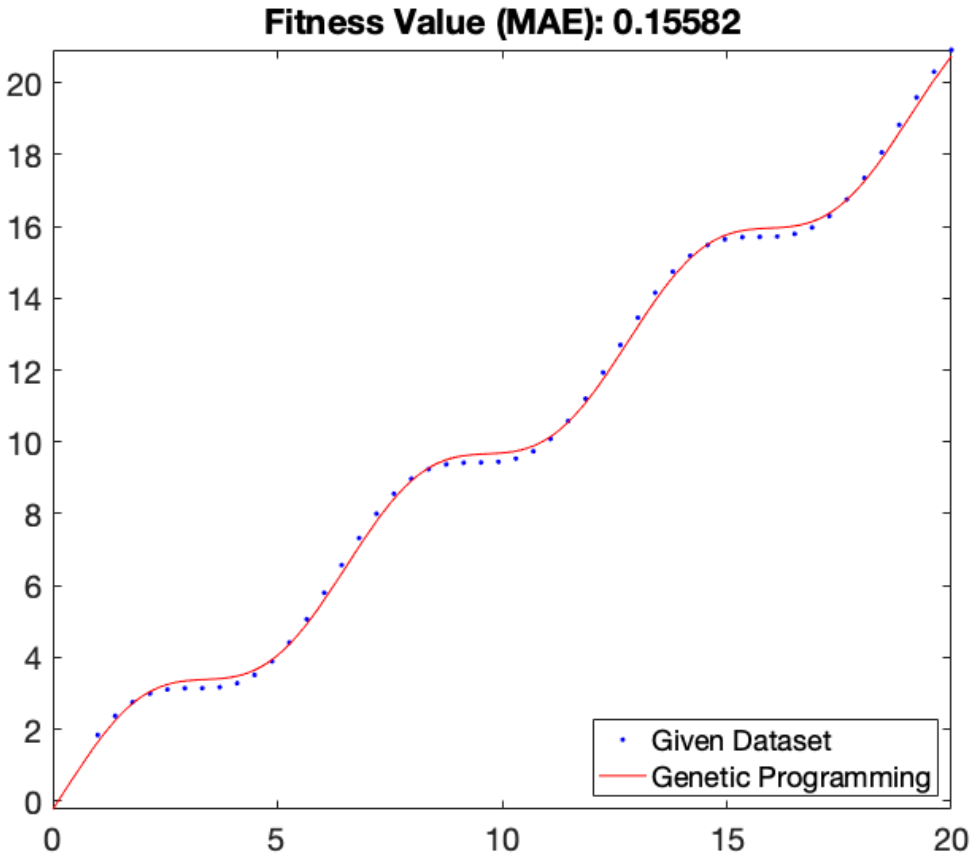

Fig 7 Test and debug with a simpler data set.

3.7 Binary Tree draw automatically

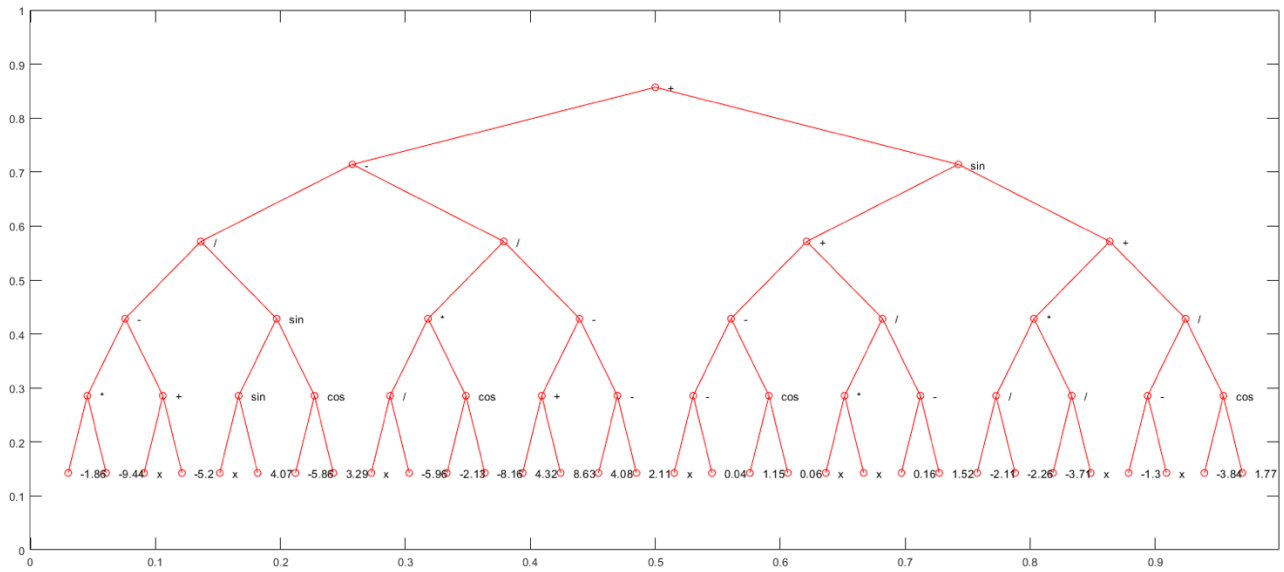

Fig 8 The binary tree with best performance created automatically by MATLAB installed function *treeplot()*. The constants in the end layer of the tree are simplified into two decimals for a clear display purpose.

3.8 Video showing the data point and best function so far

YouTube Link: https://youtu.be/qwjbaumd_pw

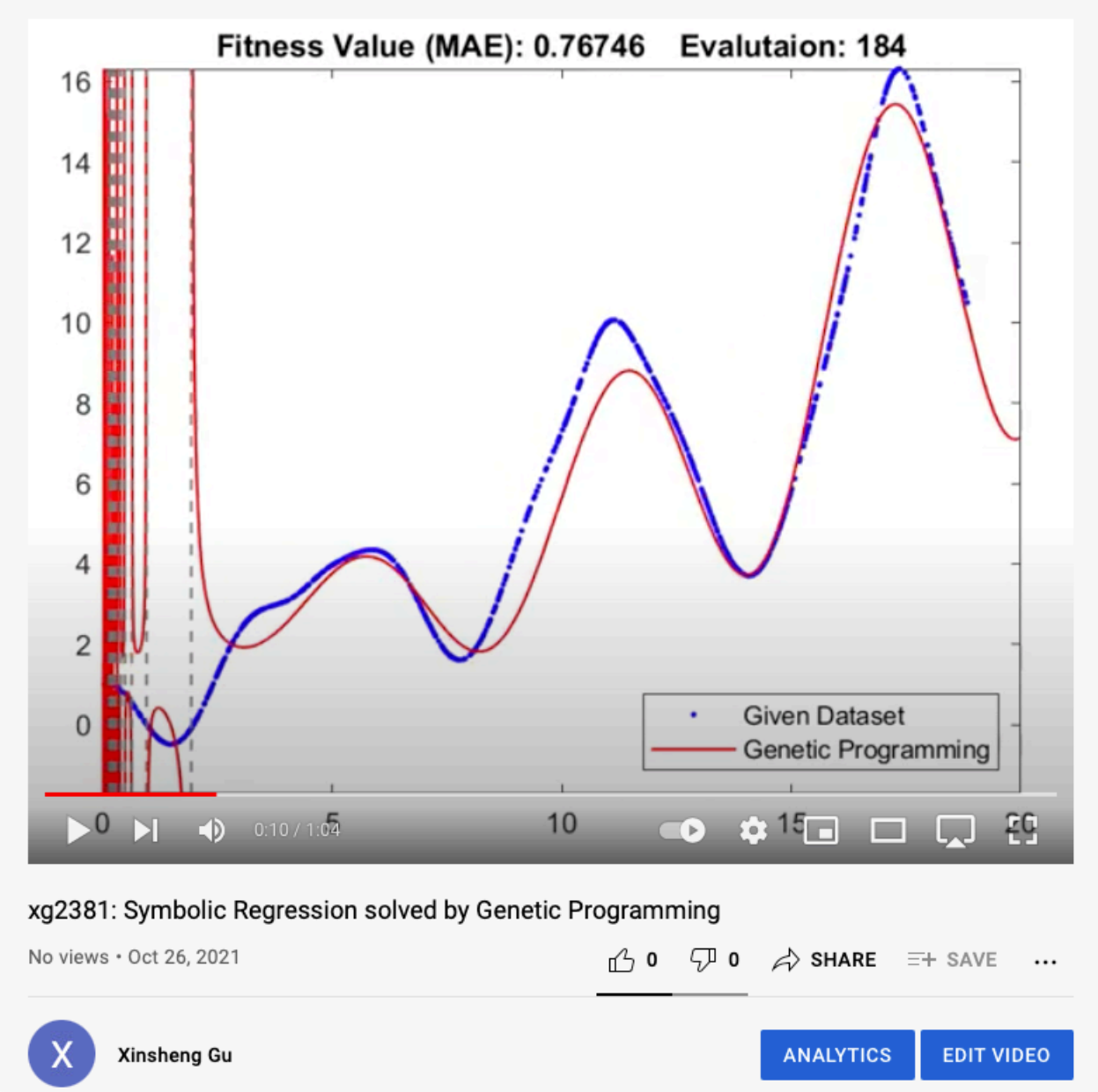

Fig 9 A snapshot of my video from YouTube

Appendix

```
%%%%%% Genetic Algorithm main %%%%%%%%
clear
close all
clc

f = importdata('data.txt');
f = f(1:10:end,:);
nRow = size(f,1);
D = 6;  % depth

Y_err_Sum = inf;
G = 100; % generation
BestList_improve = zeros(G,2^D-1);
NP = 100; % number of population
Pm = 0.08; % rate of mutation
RandomLists = zeros(NP,2^D-1);
Y_err = zeros(G,NP);
Trace = zeros(G,1);
Diversity = zeros(G,1);

for n = 1:NP
    RandomLists(n,:) = Random_create(D);
end
Lists = RandomLists;

tic
for k = 1:G
% SelectedLists = TournamentSel(Lists,f);

crossedLists = crossover(Lists);
mutatedLists = mutation(crossedLists, Pm);
% ListCombo = [Lists;mutatedLists];

[SortedListcombo, D_sorted] = SortIndividuals([Lists;mutatedLists],f);
Lists = SortedListcombo(1:round(NP/2),:);

Diversity(k) = mean(std(Lists));
Y_err(k,:) = D_sorted(1:round(NP));

if D_sorted(1) < Y_err_Sum
    Y_err_Sum = D_sorted(1); % min ErrSum recorded
    BestList_improve(k,:) = Lists(1,:);
    BestList = Lists(1,:);
    for i = 1:nRow
        [~,y_ex_Best] = TreeCal(BestList,f(i,1),f(i,2));
    end
end

Trace(k) = Y_err_Sum;
disp(['Generation:' num2str(k)])

end
Time = toc;

Fitness = -Trace/size(f,1);

%%%%%% Function Plot %%%%%%%%%%
Equ = Tree2Equ(BestList);
save('workspace.mat')

h_func = figure;
plot(f(:,1),f(:,2),'b.')
hold on
fplot(Equ,'r-','LineWidth',1)
axis([0,20,-inf,inf])
title(['Fitness Value (MAE): ' num2str(Y_err_Sum/nRow)])
legend('Given Dataset','Genetic Programming','Location','Southwest')

savefig('functionFig.fig')
```

```
saveas(h_func,'functionFig.bmp')
close

%%%%%%% Learning Curve %%%%%%%%%%
h_LC = figure;
plot(Fitness,'b-','LineWidth',1) %add a '-' in order to make the curve go up
xlabel('#Evaluation')
ylabel('Fitness Value')
title('Learning Curve for Genetic Programming')
legend('Genetic Programming')
savefig('LC.fig')
saveas(h_LC,'LC.bmp')
close

%%%%% Dot Plot with Learning Curve%%%%%%%%%%
h_dot = figure;
plot(Fitness,'b-','LineWidth',1) %add a '-' in order to make the curve go up
xlabel('#Evaluation')
ylabel('Fitness Value')
title('Dot Plot for Genetic Programming')
hold on
for i = 1:G
    for j = 1:size(Y_err,2)
    plot(i,-Y_err(i,j),'r.')
    hold on
    end
end
savefig('DotPlot.fig')
saveas(h_dot,'DotPlot.bmp')
close


%%%%%%% Convergence Plot %%%%%%%%%%
Threshold = 1;
convergence = zeros(G,1);
for i = 1:G
    count = 0;
    for j = 1:size(Y_err,2)
        if Y_err(i,j)/nRow < Threshold
            count = count + 1;
        end
    end
    convergence(i) = count/NP * 100; %Convergence(%)
end
h_cvg = figure;
plot(convergence,'b-')
xlabel('#Evaluation')
ylabel('Convergence(%)')
title(['Convergence Plot, Threshold: ' num2str(Threshold)])
axis([0,G,0,100])
savefig('ConvergencePlot.fig')
saveas(h_cvg,'ConvergencePlot.bmp')
close

%%%%%%% Diveristy Plot %%%%%%%%%%
h_div = figure;
plot(Diversity,'g*-','LineWidth',1)
xlabel('#Evaluation')
ylabel('Diversity')
title('Diversity Plot for Genetic Programming')
savefig('DiversityPlot.fig')
saveas(h_div,'DiversityPlot.bmp')
close

%%%%%%% Improvement Movie %%%%%%%%
A = all(BestList_improve == 0,2);
I = find(A == 0); % index of every improvement
for i = 1:length(I)
    handle = figure(i);
    plot(f(:,1),f(:,2),'b.')
    hold on
    Movie_Equ = Tree2Equ(BestList_improve(I(i),:));
    fplot(Movie_Equ,'r-','LineWidth',1)
```

```
    axis([0,20,-inf,inf])
    title(['Fitness Value (MAE): ' num2str(Y_err_Sum/nRow) '    Evalutaion: ' num2str(I(i))])
    legend('Given Dataset','Genetic Programming','Location','Southwest')
    saveas(handle,['Improvement' num2str(i) '.bmp'])
    close
end

------------------------------------------------------------------------------------------------
%%%%%%% Crossover %%%%%%%%
function crossedLists = crossover(Lists)

% %%%%%%% test
% clear
% close all
% clc
% load('testLists.mat')
% Lists = mutatedLists;
% NP = size(Lists,1);
% %%%%%%%

clearvars -global
NP = size(Lists,1);
% nROW = size(Lists,1); % size: NP;
nCOL = size(Lists,2); % size: 2^D - 1;
% D = log2(nCOL + 1);
crossedLists = zeros(size(Lists));

p = randperm(NP);
pairs = zeros(NP/2,2);

for i = 1:NP/2 % randomly match two parents
   pairs(i,:) = [p(2*i-1),p(2*i)];
end

for pp = 1:size(pairs,1)
    clearvars -global
    node = randi([2,nCOL]); %randomly select the crossover node
    List1 = Lists(pairs(pp,1),:);
    List2 = Lists(pairs(pp,2),:);

    CP_idx = binaryTree(List1,node); % binary Tree Recursion

    %%%%%%%% crossover %%%%%%%%%
    temp = List1(CP_idx);
    List1(CP_idx) = List2(CP_idx);
    List2(CP_idx) = temp;

    crossedLists(2*pp-1,:) = List1;
    crossedLists(2*pp,:) = List2;

end
end

function CP_idx = binaryTree(List1,node)
n = length(List1);
clear global var
global CP_idx
if node <= n
    CP_idx = [CP_idx,node];
    binaryTree(List1,2*node);
    binaryTree(List1,2*node + 1);
end
end


------------------------------------------------------------------------------------------------
%%%%%%%%% mutation for population %%%%%%%%%%
function mutatedList = mutation(List, Pm)
% input: NP*(2^D-1) matrix, where each row indicates an individual;
% input: Pm means the rate of mutation, where 0 <= Pm <= 1;
% output: mutatedList, size(NP*2^D-1);

nROW = size(List,1); % number of nRow == NP;
```

```
nCOL = size(List,2); % number of column == 2^D - 1;
D = log2(nCOL + 1); % depth of the tree;
mutatedList = List;

for i = 1:round(Pm*nROW)
H = randi(nROW); % randomly select an individual as List(H,:);

h1 = randi([2^(D-1),nCOL]); % randomly select an index of the last layer;
    g1 = rand(1);
    if g1 >= 0.3
        mutatedList(H,h1) = (rand(1)-0.5)*20;
    else
        mutatedList(H,h1) = 99;
    end
h2 = randi(2^(D-1)-1);
    mutatedList(H,h2) = random_operator();
end

end
-------------------------------------------------------------------------------------------------------
%%%%%%%%% convert numList to symList %%%%%%%%%%
function symList = numList2symList(numList)
syms x
symList = sym('void',size(numList));
    for i = 1:length(numList)
        if numList(i) == 99
            symList(i) = x;
        else
            symList(i) = numList(i);
        end
    end
end
-------------------------------------------------------------------------------------------------------
function Answer = operator_cal(oprt,a,b)
    switch oprt
        case 51
            Answer = a + b;
        case 52
            Answer = a - b;
        case 53
            Answer = a*b;
        case 54
            if b == 0 % in case zero is the denominator
                Answer = 0;
            else
                Answer = a/b;
            end
        case 61
            Answer = a*sin(b); %here sin() and cos() are work as a binary operator
        case 62
            Answer = a*cos(b);
    end



end
-------------------------------------------------------------------------------------------------------
function equ = operator_sym(oprt,a,b)
%%%% Input: sym
%%%% Output: sym
    switch oprt
        case 51
            equ = a + b;
        case 52
            equ = a - b;
        case 53
            equ = a*b;
        case 54
            equ = a/b;
        case 61
            equ = a*sin(b);
        case 62
            equ = a*cos(b);
```

```
    end
end

------------------------------------------------------------------------------------------------
function List = Random_create(D) % D: depth of tree
L = 2^D - 1; %L: length of array
List = zeros(1,L); %create a list consists of 'num' entirely

for i = 2^(D-1):L
    h1 = rand(1);
    if h1 >= 0.3
        List(i) = (rand(1)-0.5)*20;
    else
        List(i) = 99;
    end
end

for i = 1:(2^(D-1)-1)
    List(i) = random_operator();
end

end


------------------------------------------------------------------------------------------------=
function operator = random_operator()
    k = randi(6);
        switch k
            case 1
                operator = uint8(51);
            case 2
                operator = uint8(52);
            case 3
                operator = uint8(53);
            case 4
                operator = uint8(54);
            case 5
                operator = uint8(61);
            case 6
                operator = uint8(62);
        end


end
------------------------------------------------------------------------------------------------
%%%%%%% Sorted fitness %%%%%%%
function [SortedList, D_sorted] = SortIndividuals(Lists,f) %List: NP*(2^D-1) Matrix
%%%%% sort the population in each generation %%%%%
y_err = zeros(size(f,1),1); % y-value error for each datapoint in one iteration
y_ex = zeros(size(f,1),1); % list of y-value for every iteration

N = size(Lists,1);
D = zeros(N,1);
for n = 1:N
    for i = 1:size(f,1)
        [y_err(i),y_ex(i)] = TreeCal(Lists(n,:),f(i,1),f(i,2));
    end
D(n) = sum(y_err);
end

[D_sorted,I] = sort(D,'ascend'); %finding the minimum
SortedList = Lists(I,:); % the sorted Lists matrix

end

------------------------------------------------------------------------------------------------
%%%%%%% From Binary Tree to Readable equation %%%%%%%%
function Equ = Tree2Equ(List) %Input: num list

L = size(List,2); %calculate length of List
syms x
symList = numList2symList(List);
```

```
i = L;
while i >= 3
    equ = operator_sym(symList((i-1)/2),symList(i),symList(i - 1));
    symList((i-1)/2) = equ;
    symList(i) = [];
    symList(i-1) = [];
    i = i - 2;
end
Equ = symList;

End

-------------------------------------------------------------------------------------------------------
function [y_err,y_ex] = TreeCal(List,xdata,ydata) %List of heap, xdata: num, ydata num


L = size(List,2); %calculate length of List

xvalue = xdata; % given a x value
    for i = 1:L
        if List(i) == 99
            List(i) = xvalue;
        end
    end


i = L;
while i >= 3
    Answer = operator_cal(List((i-1)/2),List(i),List(i - 1));
    List((i-1)/2) = Answer;
    List(i) = [];
    List(i-1) = [];
    i = i - 2;
end
y_ex = List(i);
y_err = abs(ydata - List(i));

end
-------------------------------------------------------------------------------------------------------
%%%% draw binary tree automatically %%%%%
function h_BTDA = BinaryTreeDrawAuto(List)

h_BTDA = figure;
strList = numList2strList(List);
D = log2(length(List) + 1);

nodes = [0,reshape([[1:1:2^(D-1)-1];[1:1:2^(D-1)-1]],[1,2*(2^(D-1)-1)])];
treeplot(nodes)

[x,y] = treelayout(nodes);
offset = 0.01;
for i = 1:length(x)
    text(x(i)+offset, y(i),strList(i))
end

-------------------------------------------------------------------------------------------------------
%%%%%%%%% List to str %%%%%%%%%
%%%%% used for drawing binary tree automatically %%%%%%%
function strList = numList2strList(List)

strList = string(zeros(size(List)));

for i = 1:length(List)
    if List(i) > 50
        switch List(i)
            case 51
               strList(i) = '+';
            case 52
               strList(i) = '-';
            case 53
               strList(i) = '*';
            case 54
               strList(i) = '/';
```

```
            case 61
               strList(i) = 'sin';
            case 62
               strList(i) = 'cos';
            case 99
                strList(i) = 'x';
        end
    else
        strList(i) = num2str(List(i));
    end
end
```