COLUMBIA UNIVERSITY

MECE 4510 EVOLUTIONARY COMPUTATION AND DESIGN AUTOMATION

Symbolic Regression

Hanwen Zhao
UNI: hz2547

supervised by
Dr. Hod LIPSON

Grace Hours Used: 53
Grace Hours Remaining: 43

October 22, 2018

1 Result Summary

Result Table		
Method	Evaluation Number	Best Error (MAE)
Random Search	300000	0.0847
Hill Climber	300000	0.0863
GP (Conventional Selection)	14000	0.00002537
GP (Conventional Selection) with Larger Population	43000	0.00002537
GP (Deterministic Crowding)	129000	0.00002537

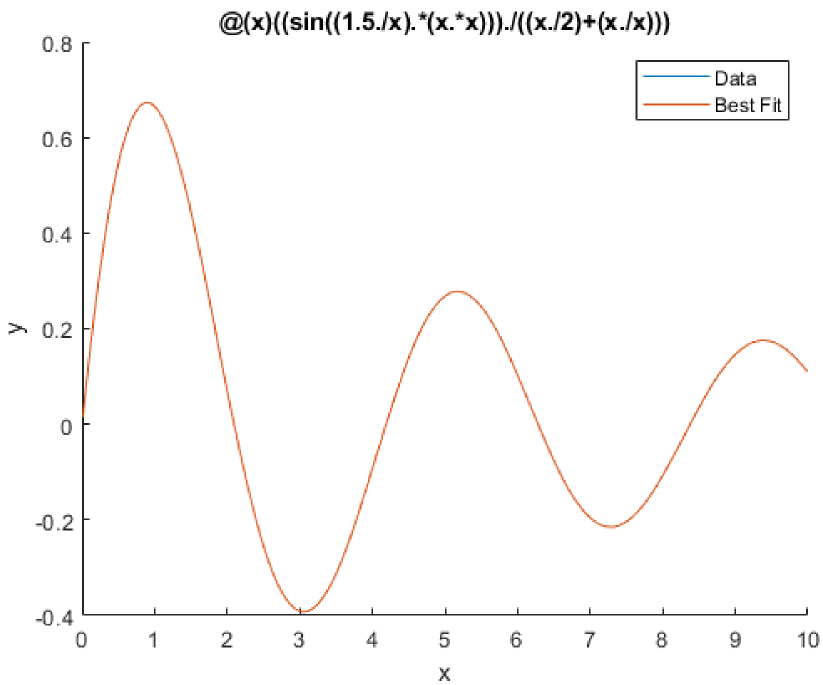


Figure 1: Performance Plot

After simplification, the function can be expressed as:

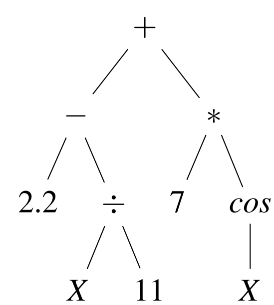
$$f(x) = sin(1.5x)/(0.5x+1) \tag{1}$$

2 Methods

For this homework, we have been asked to use genetic programming to perform symbolic regression. Our goal is to find the symbolic algebraic expression in the form of $y = f(x)$ that best fits a set of given 1000 (x, y) points. Assume the symbolic regression only uses algebraic operators: x, $-$, $*$, $/$, *sine and cosine*, and real constants(in the range ± 10), and the variable x.

2.1 Representation

The key part of genetic programming is to covert the program into a high level tree structure compare to genetic programming which the program was converted into simpler chromosome type. The tree structure is more powerful in terms of computer programming since trees can be easy evaluated in a recursive manner. For example, a function can be represented as a tree structure as following:



can represent the equation

$$(2.2 - \frac{x}{11}) + (7 * cos(x)) \tag{2}$$

For this assignment, we assume the maximum depth of the tree structure is less and equal than 5.

2.2 Random Search

Once we have our representation setup, the random search algorithm is straight forward. For each evaluation of the random search algorithm, a function tree is generated with random depth from two to five. The random search is the baseline for performance comparison between the hill climber algorithm and our genetic programmings.

2.3 Random Hill Climber

A Random Hill Climber is basically a random search with simple decision making capability. Between each evaluation, the mutation process is applied to the function. During the mutation process, it will randomly select a valid mutation point, generate a new sub-tree and replace into new function tree. In my implementation, the probability decide whether it always go to a better solution or not. Sometime go to a worse solution can avoid hill climber stuck at the local maximum.

2.4 Genetic Programming with Variations

Similar to Evolutionary Algorithm, we can implement following types of operators into the Genetic Programming: selection, crossover, and mutation. For variation, Deterministic Crowding and different size of population was applied during my implementation. Here are some details about the techniques I used in my implementation:

- Selection: For conventional selection Genetic Programming, during each generation, a default of 50% parents will be selected to generate 50 offsprings.
- Crossover: The crossover was applied into both conventional selection method and deterministic crowding method. The algorithm will first choose two valid crossover node from each parent and switch the subtrees between two parents in order to generate two offSprings.
- Mutation: During the mutation process, the algorithm will first randomly pick a valid node for mutation, then generate a new tree based on the depth of the mutation node.
- Deterministic Crowding: In order to maintain the diversity for genetic programming, we need some methods to maintain useful diversity for genetic programming to work better. Crowding is one of the popular method, it only replace individuals that are similar. More specific, deterministic crowding compare the similarity between two parents and two offsrpings, replace the one has higher similarity.

Algorithm 1 Deterministic Crowding

procedure MYPROCEDURE
    if $d(p_1, c_1) + d(p_2, c_2) < d(p_1, c_2) + d(p_2, c_1)$ **then**
        compare c_1 *to* p_1 *and* c_2 *to* p_2 *and replace parents if offspring better*
    else
        compare c_1 *to* p_2 *and* c_2 *to* p_1 *and replace parents if offspring better*

2.5 Analysis of Performance

Overall, all GPs performed very well since they all can find the optimal solution. However, the random search and hill climber did not perform as expected. The main reason for the failure of hill climber because it was badly guided. Like random search, hill climber would perform well if it is lucky. On the other hand, the deterministic crowding for genetic programming is extremely powerful. From both performance plot as well as the diversity plot, the crowding helped to maintain diversity in a better manner compare to conventional selection. Therefore, on the performance plot, the crowding starts slow than conventional selection, but it could find better solution eventually.

3 Performance

3.1 Performance Plots

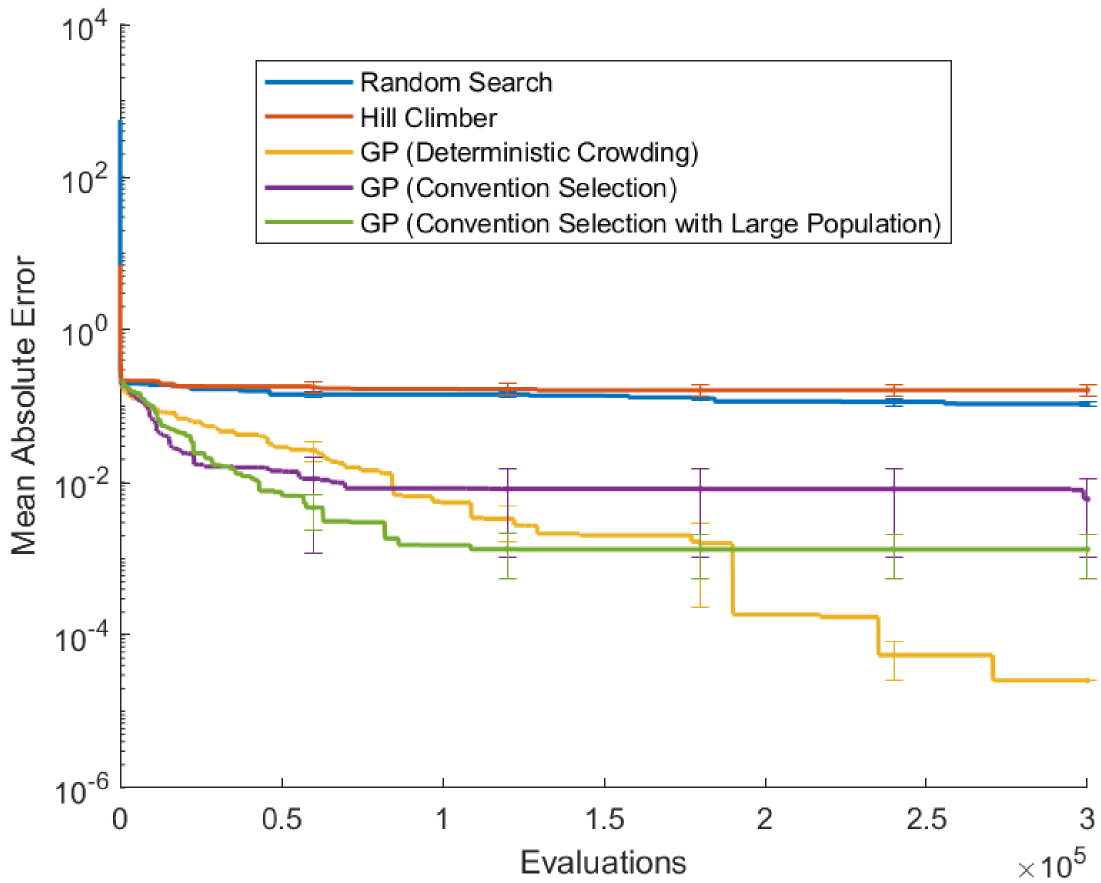


Figure 2: Performance Plot

3.2 Dot Plot

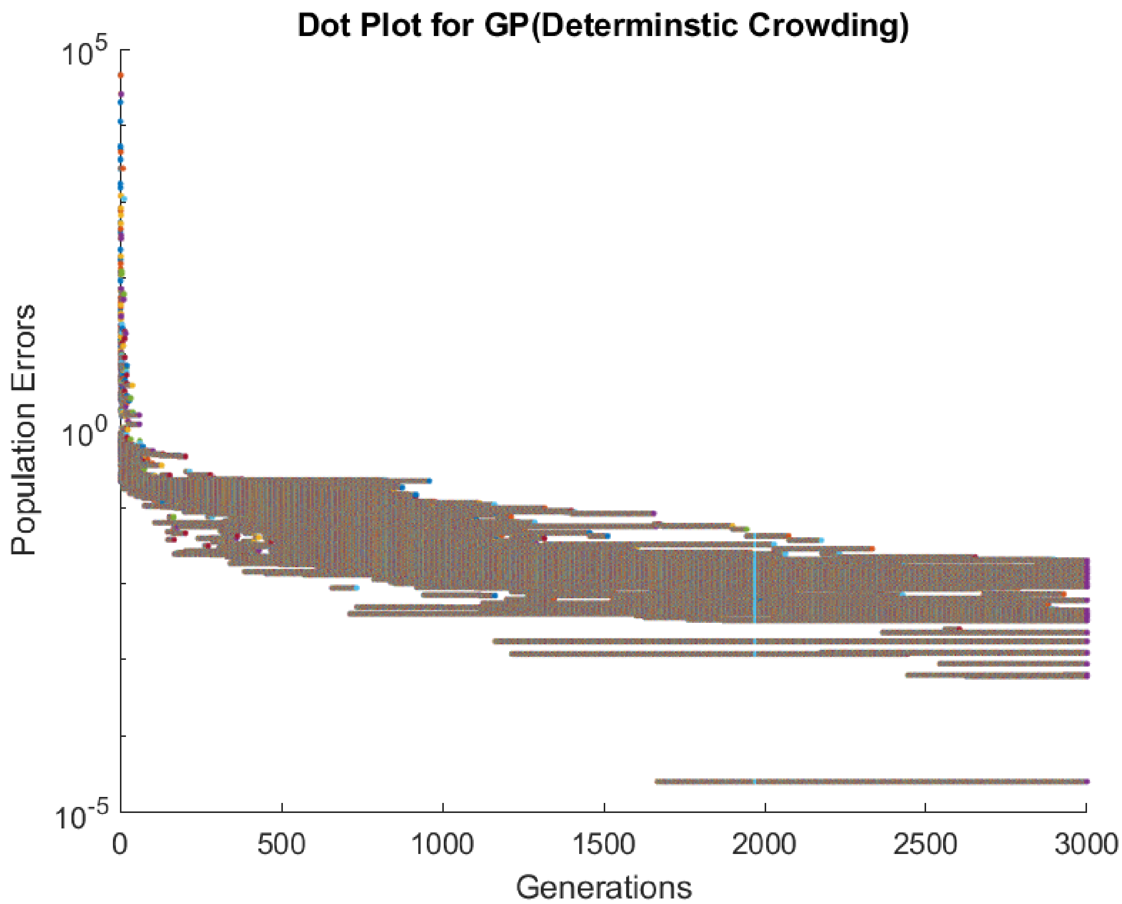


Figure 3: Dot Plot

3.3 Diversity Plot

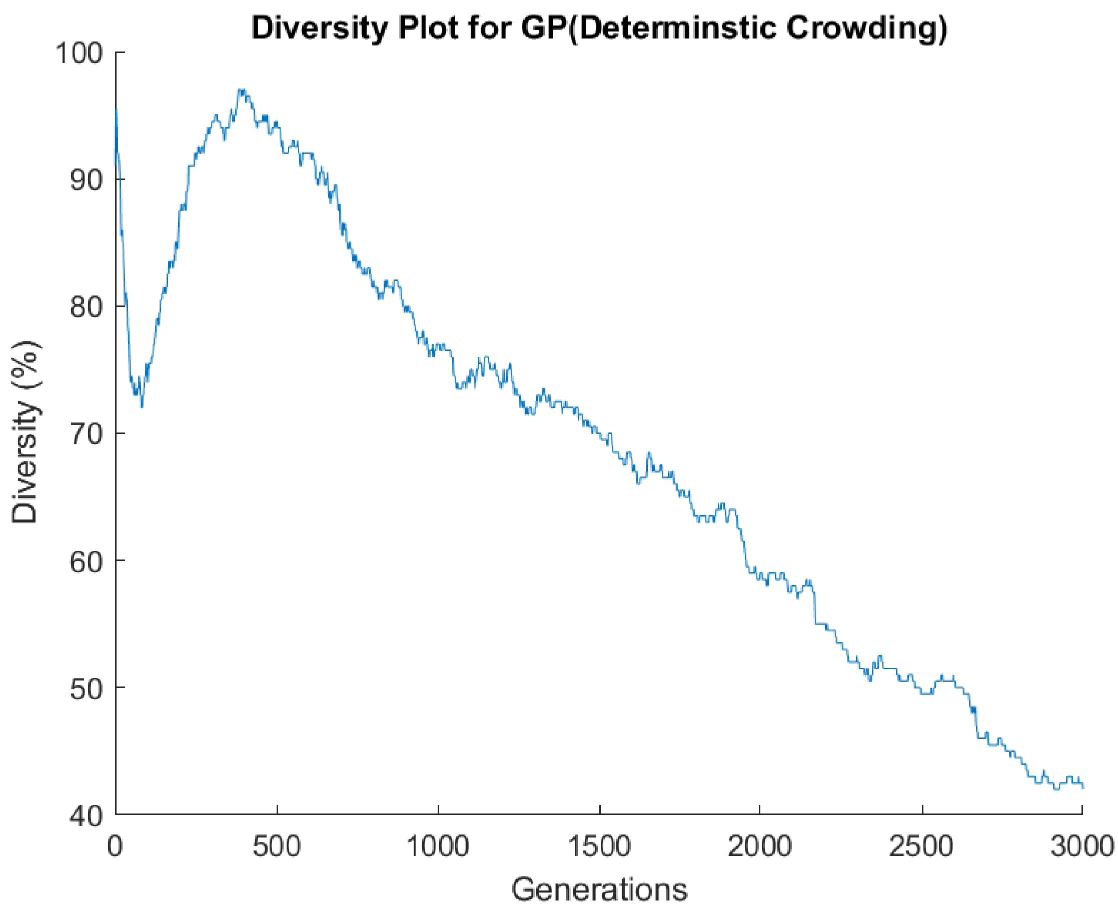


Figure 4: Diversity Plot

3.4 Convergence Plot

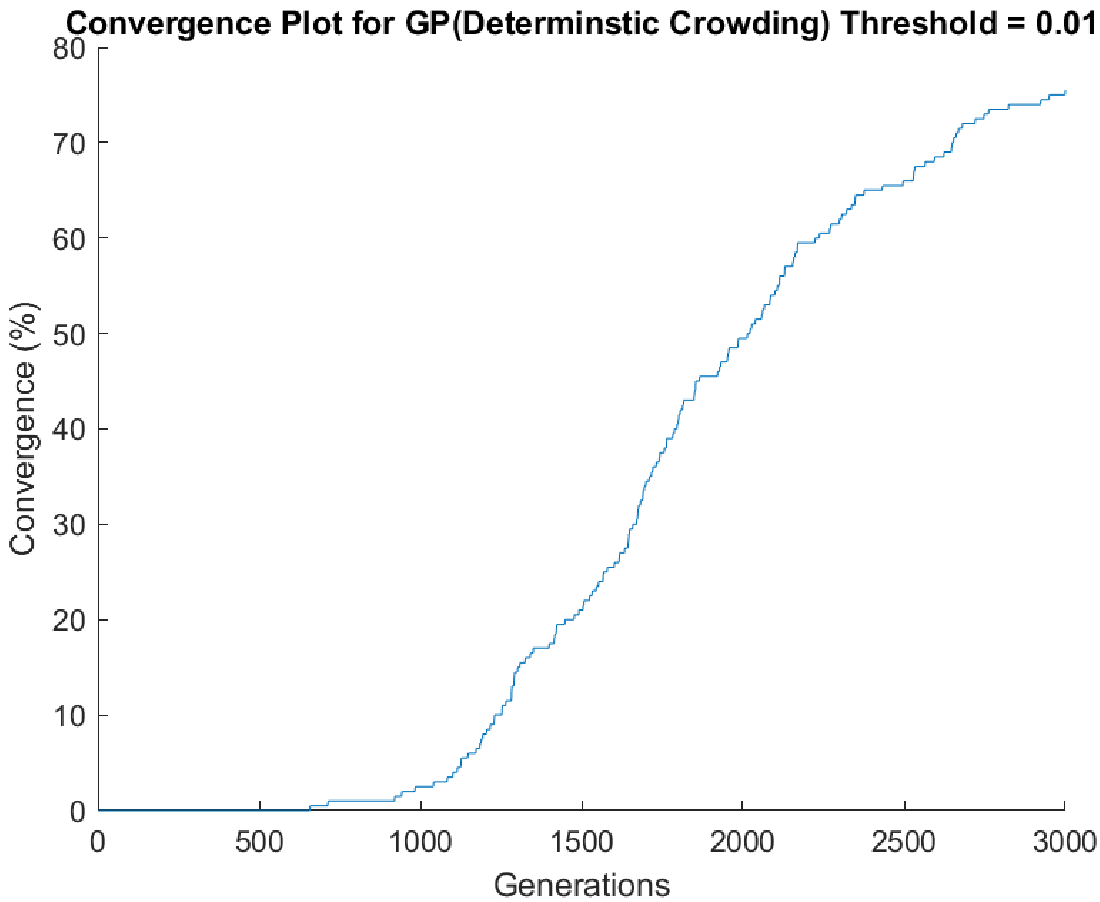


Figure 5: Convergence Plot

3.5 Simpler Problem Tested

During the debugging process of Genetic Programming, simpler problem was tested to ensure the GP can run without any bug. For example, a test data set of $sin(x)$ was used. In addition, all GPs can find the optimal solution within 10 generations.

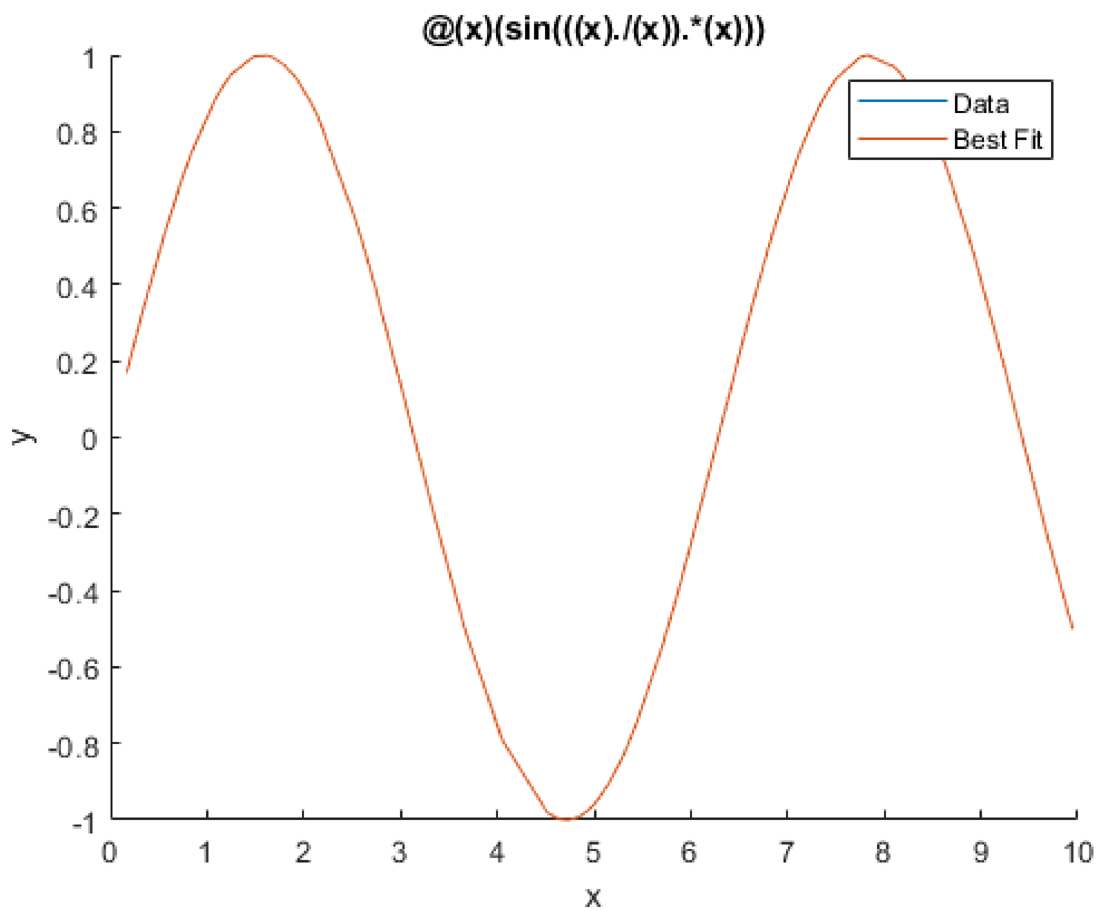


Figure 6: Simple Problem Test

3.6 Validation

The validation plot did not show much difference between training data and testing data since the data we were given has negligible noise.

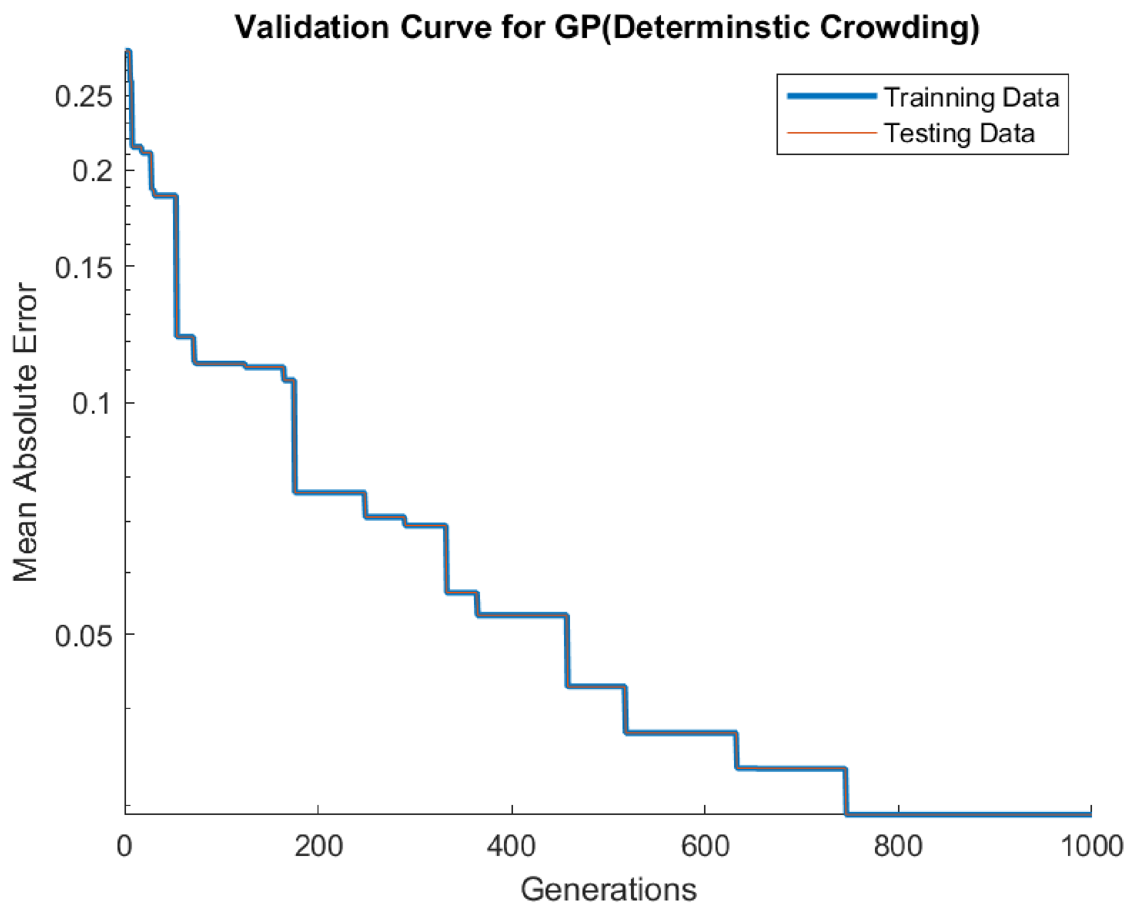


Figure 7: Validation Plot

4 Appendix

```
function [] = HW2()
operator = {'+','-','.*','./','sin','cos'};
varConst = {'c','x'};
maxLevel = 4;
tic
rng('shuffle')
data = csvread('function1.csv');
%trainingIndex = sort(randperm(1000, 100));
trainingIndex = 1:500;
trainingData = data(trainingIndex,:);
validationData = data(setdiff(1:length(data),trainingIndex),:);
randomSearch(operator, varConst, maxLevel, trainingData, 300000, 4);
hillClimber(operator, varConst, maxLevel, trainingData, 300000, 4);
GP1(operator, varConst, maxLevel, trainingData, validationData, 200, 3000, 4, 'true', 'dot');
GP2(operator, varConst, maxLevel, trainingData, 200, 3000, 4, 'true');
GP2_LP(operator, varConst, maxLevel, trainingData, 400, 1500, 4, 'true');
toc
end

function [] = GP2_LP(operator, varConst, maxLevel, data, popSize, n, repeat, print)
parfor r = 1:repeat
    if strcmp(print,'true')
        fileID = fopen(strcat('GP2_LP_',int2str(r),'.txt'),'wt');
    end
    population = cell([popSize,2^maxLevel]);
    for ii = 1:popSize
        population(ii,1:end-1) = heapGeneration(operator, varConst, maxLevel);
    end
    popError = zeros(popSize,1);
    bestError = inf;
    for i = 1:n
       for ii = 1:popSize
           population{ii,2^maxLevel} = MAE_Cal(population(ii,1:end-1),data);
           popError(ii) = population{ii,2^maxLevel};
           %heapString(population(ii,1:end-1))
       end
       currentError = min(popError);
       population = sortrows(population,2^maxLevel);
       selectPop = population(1:popSize/2,1:end-1);
       offSpringPop = cell([popSize/2,2^maxLevel-1]);
       bestError = min(currentError,bestError);
       if strcmp(print,'true')
           fprintf(fileID, '%d %10.8f %10.8f \n', i*popSize/2, currentError, bestError);
       end
       for jj = 1:popSize/4
           parent1 = selectPop(randi(length(selectPop)),:);
           parent2 = selectPop(randi(length(selectPop)),:);
           [offSpring1, offSpring2] = crossOver(parent1, parent2, maxLevel);
           offSpring1 = mutation(offSpring1, operator, varConst, maxLevel);
           offSpring2 = mutation(offSpring2, operator, varConst, maxLevel);
           offSpringPop(jj*2-1:jj*2,:) = [offSpring1;offSpring2];
       end
       population(:,1:end-1) = [selectPop;offSpringPop];
    end
    population = population(:,1:end-1);
    for i = 1:popSize
        popError(i) = MAE_Cal(population(i,:),data);
    end
%     bestHeap = population(1,:);
%     bestHeapString = heapString(bestHeap);
%     figure
%     hold on
%     plot(data(:,1),data(:,2))
%     plot(data(:,1),evalHeap(bestHeap, data))
%     legend('Data','Best Fit')
```

```
%     xlabel('x')
%     ylabel('y')
%     title(bestHeapString)
    index = find(popError == min(popError));
    fprintf("GP2: Eval: %d, Error: %10.8f.\n", n, popError(index(1)))
end
end

function [] = GP2(operator, varConst, maxLevel, data, popSize, n, repeat, print)
parfor r = 1:repeat
    if strcmp(print,'true')
        fileID = fopen(strcat('GP2_',int2str(r),'.txt'),'wt');
    end
    population = cell([popSize,2^maxLevel]);
    for ii = 1:popSize
        population(ii,1:end-1) = heapGeneration(operator, varConst, maxLevel);
    end
    popError = zeros(popSize,1);
    bestError = inf;
    for i = 1:n
       for ii = 1:popSize
           population{ii,2^maxLevel} = MAE_Cal(population(ii,1:end-1),data);
           popError(ii) = population{ii,2^maxLevel};
           %heapString(population(ii,1:end-1))
       end
       currentError = min(popError);
       population = sortrows(population,2^maxLevel);
       selectPop = population(1:popSize/2,1:end-1);
       offSpringPop = cell([popSize/2,2^maxLevel-1]);
       bestError = min(currentError,bestError);
       if strcmp(print,'true')
           fprintf(fileID, '%d %10.8f %10.8f \n', i*popSize/2, currentError, bestError);
       end
       for jj = 1:popSize/4
           parent1 = selectPop(randi(length(selectPop)),:);
           parent2 = selectPop(randi(length(selectPop)),:);
           [offSpring1, offSpring2] = crossOver(parent1, parent2, maxLevel);
           offSpring1 = mutation(offSpring1, operator, varConst, maxLevel);
           offSpring2 = mutation(offSpring2, operator, varConst, maxLevel);
           offSpringPop(jj*2-1:jj*2,:) = [offSpring1;offSpring2];
       end
       population(:,1:end-1) = [selectPop;offSpringPop];
    end
    population = population(:,1:end-1);
    for i = 1:popSize
        popError(i) = MAE_Cal(population(i,:),data);
    end
%     bestHeap = population(1,:);
%     bestHeapString = heapString(bestHeap);
%     figure
%     hold on
%     plot(data(:,1),data(:,2))
%     plot(data(:,1),evalHeap(bestHeap, data))
%     legend('Data','Best Fit')
%     xlabel('x')
%     ylabel('y')
%     title(bestHeapString)
    index = find(popError == min(popError));
    fprintf("GP2: Eval: %d, Error: %10.8f.\n", n, popError(index(1)))
end
end

function [] = GP1(operator, varConst, maxLevel, data, validationData, popSize, n, repeat, print,
    type)
dotData = zeros(n,popSize);
errorAndValidation = zeros(n,2);
for r = 1:repeat
    if strcmp(print,'true')
```

```
        fileID = fopen(strcat('GP1_',int2str(r),'.txt'),'wt');
    end
    population = cell([popSize,2^maxLevel-1]);
    for ii = 1:popSize
        population(ii,:) = heapGeneration(operator, varConst, maxLevel);
    end
    popError = zeros(popSize,1);
    validationError = zeros(popSize,1);
    bestError = inf;
    for i = 1:n
        for ii = 1:popSize
           popError(ii) = MAE_Cal(population(ii,:),data);
           validationError(ii) = MAE_Cal(population(ii,:),validationData);
           %heapString(population(ii,:))
        end
        mean(popError,'omitnan');
        currentError = sort(popError);
        validationError = sort(validationError);
        bestError = min(currentError(1),bestError);
        errorAndValidation(i,:) = [currentError(1) validationError(1)];
        if strcmp(print,'true')
            fprintf(fileID, '%d %10.8f %10.8f \n', i*popSize/2, currentError(1), bestError);
        end
        if strcmp(type,'dot')
            dotData(i,:) = popError;
        end
        % random select two parents
        for j = 1:popSize/2
            index = randperm(popSize,2);
            parent1 = population(index(1),:);
            parent2 = population(index(2),:);
            % crossover to get two offsprings
            [offSpring1, offSpring2] = crossOver(parent1, parent2, maxLevel);
            % perfome mutation
            offSpring1 = mutation(offSpring1, operator, varConst, maxLevel);
            offSpring2 = mutation(offSpring2, operator, varConst, maxLevel);
            % perfome deterministic crowding
            dp1c1 = similarCal(parent1, offSpring1, data);
            dp2c2 = similarCal(parent2, offSpring2, data);
            dp1c2 = similarCal(parent1, offSpring2, data);
            dp2c1 = similarCal(parent2, offSpring1, data);
            c1 = MAE_Cal(offSpring1, data); p1 = MAE_Cal(parent1, data);
            c2 = MAE_Cal(offSpring2, data); p2 = MAE_Cal(parent2, data);
            if dp1c1+dp2c2 < dp1c2+dp2c1
               if c1 < p1
                   population(index(1),:) = offSpring1;
               end
               if c2 < p2
                   population(index(2),:) = offSpring2;
               end
            else
                if c1 < p2
                    population(index(2),:) = offSpring1;
                end
                if c2 < p1
                    population(index(1),:) = offSpring2;
                end
            end
        end
    end
    for i = 1:popSize
        popError(i) = MAE_Cal(population(i,:),data);
    end
%     sortError = sort(popError);
%     [~,errorIndex] = ismember(sortError,popError);
%     sortPop = population(errorIndex,:);
%     bestHeap = sortPop(1,:);
%     bestHeapString = heapString(bestHeap);
```

```
%      figure
%      hold on
%      plot(data(:,1),data(:,2))
%      plot(data(:,1),evalHeap(bestHeap, data))
%      legend('Data','Best Fit')
%      xlabel('x')
%      ylabel('y')
%      title(bestHeapString)
    index = find(popError == min(popError));
    fprintf("GP1: Eval: %d, Error: %10.8f.\n", n, popError(index(1)))
    save('dotData','dotData')
    save('errorAndValidation2','errorAndValidation')
end
end

function sim = similarCal(heap1, heap2, data)
sim = abs(MAE_Cal(heap1,data)-MAE_Cal(heap2,data));
end

function [offSpring1, offSpring2] = crossOver(parent1, parent2, maxLevel)
maxCrossOverLimit = min(max(find(~cellfun(@isempty,parent1))),max(find(~cellfun(@isempty,parent2)
    )));
offSpring1 = parent1;
offSpring2 = parent2;
maxCrossOverLevel = floor(log2(maxCrossOverLimit))+1;
while 1
crossOverLevel = randi(maxCrossOverLevel);
selection = 2^(crossOverLevel-1):2^(crossOverLevel)-1;
crossOverPoint1 = selection(randi(length(selection)));
crossOverPoint2 = selection(randi(length(selection)));
if ~isempty(parent1{crossOverPoint1}) && ~isempty(parent2{crossOverPoint2})
    if isnumeric(parent1{crossOverPoint1}) && isnumeric(parent2{crossOverPoint2})
        break
    elseif strcmp(parent1{crossOverPoint1},'x') && strcmp(parent2{crossOverPoint2},'x')
        break
    elseif isnumeric(parent1{crossOverPoint1}) && strcmp(parent2{crossOverPoint2},'x')
        break
    elseif isnumeric(parent2{crossOverPoint2}) && strcmp(parent1{crossOverPoint1},'x')
        break
    elseif ischar(parent1{crossOverPoint1}) && ischar(parent2{crossOverPoint2})
        break
    end
end
end
crossOverLocations1 = searchChildren(crossOverPoint1, maxLevel);
crossOverLocations2 = searchChildren(crossOverPoint2, maxLevel);
subHeap1 = parent1(crossOverLocations1);
subHeap2 = parent2(crossOverLocations2);
for i = 1:length(crossOverLocations1)
    offSpring1(crossOverLocations1(i)) = subHeap2(i);
    offSpring2(crossOverLocations2(i)) = subHeap1(i);
end
end

function [] = hillClimber(operator, varConst, maxLevel, data, n, repeat)
parfor r = 1:repeat
    fileID = fopen(strcat('RMHC_',int2str(r),'.txt'),'wt');
    heap = heapGeneration(operator, varConst, maxLevel);
    oldError = MAE_Cal(heap,data);
    bestError = inf;
    for i = 1:n
       newHeap = mutation(heap, operator, varConst, maxLevel);
       newError = MAE_Cal(newHeap, data);
       if newError < oldError
          heap = newHeap;
          oldError = newError;
          bestError = newError
       end
```

```
       fprintf(fileID, '%d %10.8f %10.8f \n', i, newError, bestError);
    end
    fprintf("RMHC: Eval: %d, Error: %10.8f.\n", n, bestError)
end
end

function heap = mutation(heap, operator, varConst, maxLevel)
% the mutation should be able happen non-empty node
mutateIndex = find(~cellfun(@isempty,heap));
mutationPoint = mutateIndex(randi(length(mutateIndex)-1)+1);
mutateLocations = searchChildren(mutationPoint, maxLevel);
% determine the mutation level
mutationLevel = floor(log2(mutationPoint))+1;
% delete the original heap nodes
for i = 1:length(mutateLocations)
heap{mutateLocations(i)} = [];
end
if mutationLevel == maxLevel
    heap(mutateLocations) = varConst(randi(length(varConst)));
else
    % generate a subtree
    subHeap = heapGeneration(operator, varConst, (maxLevel-mutationLevel+1));
    for i = 1:length(mutateLocations)
       heap(mutateLocations(i)) = subHeap(i);
    end
end
% replace c with constant
heap = replaceC(heap);
end

function operateLocations = searchChildren(operatPoint, maxLevel)
searchQueue = [operatPoint];
operateLocations = [operatPoint];
while ~isempty(searchQueue)
    currentIndex = searchQueue(1);
    if currentIndex*2+1 <= 2^maxLevel - 1
        operateLocations(end+1) = currentIndex*2;
        operateLocations(end+1) = currentIndex*2+1;
        searchQueue(end+1) = currentIndex*2;
        searchQueue(end+1) = currentIndex*2+1;
    end
    searchQueue(1) = [];
end
end

function [] = randomSearch(operator, varConst, maxLevel, data, n, repeat)
parfor r = 1:repeat
    fileID = fopen(strcat('Random_',int2str(r),'.txt'),'wt');
    bestError = inf;
    %bestHeap = cell([2^maxLevel-1,1]);
    for i = 1:n
        % maximum level of heap can vary from 2 to 4
        heap = heapGeneration(operator, varConst, maxLevel);
        error = MAE_Cal(heap, data);
        if error < bestError
            bestError = error
            %bestHeap = heap;
        end
        fprintf(fileID, '%d %10.8f %10.8f \n', i, error, bestError);
    end
    fprintf("RM: Eval: %d, Error: %10.8f.\n", n, bestError)
end
end

function eval = evalHeap(heap, data)
heapStr = heapString(heap);
fh = str2func(heapStr);
eval = fh(data(:,1));
```

```
end

function error = MAE_Cal(heap, data)
heapStr = heapString(heap);
fh = str2func(heapStr);
y = fh(data(:,1));
error = sum(abs(data(:,2) - y))/length(y);
end

function heap = heapGeneration(operator, varConst, maxLevel)
heapSize = 2^maxLevel - 1;
heap = cell([heapSize,1]);
opSize = 2^(maxLevel-2)-1;
operatorQueue = [1];
while ~isempty(operatorQueue)
    % pick the current index
    currentIndex = operatorQueue(1);
    % assign current operator
    heap(currentIndex) = operator(randi(length(operator)));
    % make sure the operator assignment does not excced the limit
    if currentIndex <= opSize
       if rand < 0.5
          operatorQueue(end+1) = currentIndex*2;
       end
       if rand < 0.5
           operatorQueue(end+1) = currentIndex*2+1;
       end
    end
    % delete the first in queue
    operatorQueue(1) = [];
end
% record the operator index
opIndex = find(~cellfun(@isempty,heap));
varConstIndex = [];
for i = 1:length(opIndex)
    if isempty(heap{opIndex(i)*2})
       varConstIndex(end+1) = opIndex(i)*2;
    end
    if isempty(heap{opIndex(i)*2+1})
       varConstIndex(end+1) = opIndex(i)*2+1;
    end
end
% assign x and c to the rest of the tree
for i = 1:length(varConstIndex)
    heap(varConstIndex(i)) = varConst(randi(length(varConst)));
end
% replace the 'c' with acutal constant
heap = replaceC(heap);
end

function heap = replaceC(heap)
heapSize = length(heap);
for i = 1:heapSize
    const = -10:0.1:10;
    if strcmp(heap{i},'c')
       heap{i} = const(randi(length(const)));
    end
end
end

function heapStr = heapString(heap)
heapStr = heap;
% reverse order from 15 to 1
for i = fliplr(1:floor(length(heap)/2))
    % make sure current node is not empty
    if ~isempty(heapStr{i})
        % if current node is sin or cos, only combine its right child
        if strcmp(heapStr{i},'sin') || strcmp(heapStr{i},'cos')
```

```
            %heapStr{i} = strcat(num2str(heapStr{i*2}),'*',num2str(heapStr{i}),'(',num2str(heapStr{2*i+1}),')');
            %heapStr{i} = strcat(num2str(heapStr{i}),'(',num2str(heapStr{2*i+1}),')');
            heapStr{i} = strcat('(',num2str(heapStr{i}),'(',num2str(heapStr{2*i}),'.*',num2str(heapStr{2*i+1}),')',')');
        else
            % else combine the left child, current node and right child
            heapStr{i} = strcat('(',num2str(heapStr{2*i}),num2str(heapStr{i}),num2str(heapStr{2*i+1}),')');
        end
    end
end
% take the top of the heap as output
heapStr = strcat('@(x) ',heapStr{1});
end
```