Genetic Symbolic Regression

Name: Huy Ha
UNI: hqh2101

Course Name: Evolutionary Computation and Automated Design
Course Number: MECS E4510
Instructor: professor Hod Lipson

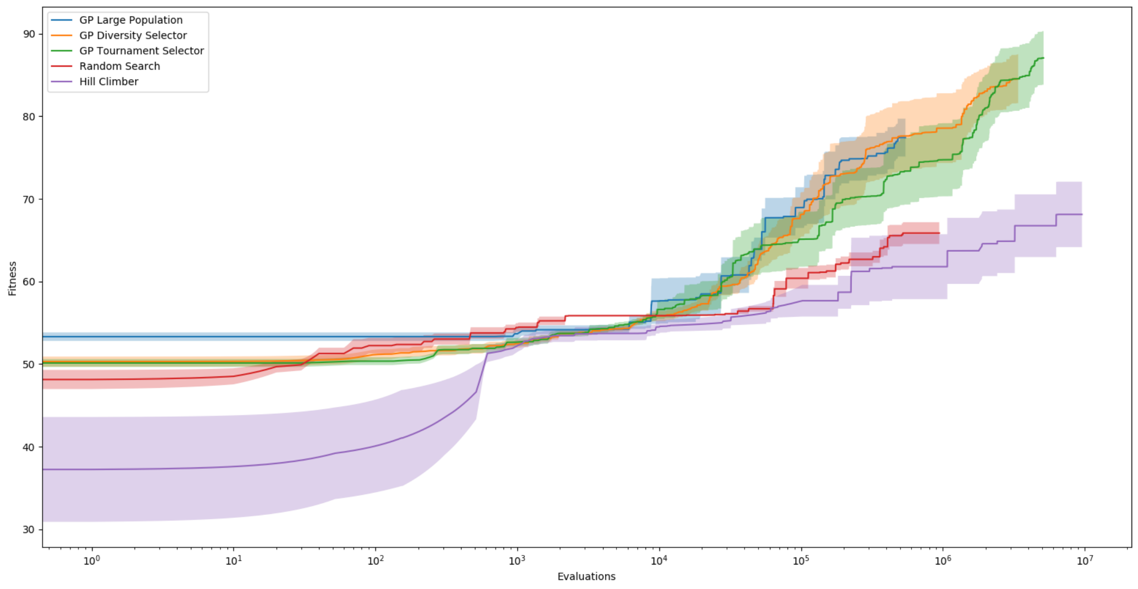
Date Submitted: October 22th 2019
Grace Hours used: 46 hours
Grace Hours left: 50 hours

1. Results Page

1.1. Results summary table

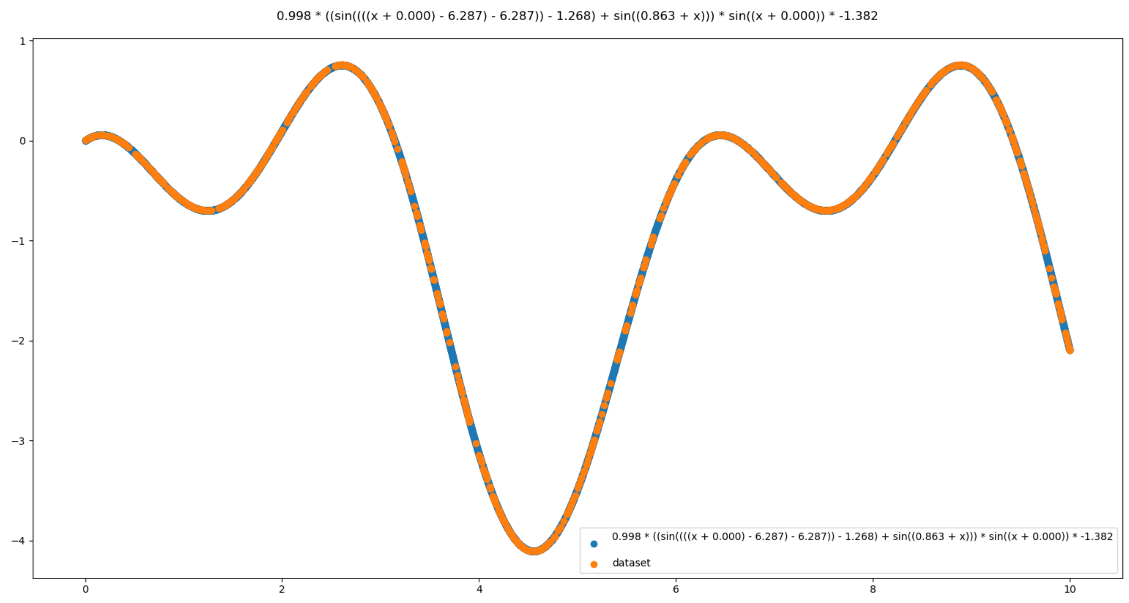
Method	Configuration	Average	Best
Random Search	N/a	66.40	69.80
Hill Climber	N/a	68.23	82.65
Base GP	Population size 50, Diversity Selector	87.66	98.27
Tournament GP	Population size 50, Tournament Selector	93.3	99.90
Large Population GP	Population size 500, Diversity Selector	82.19	88.72

1.2. Learning Curves of methods tried with Error bars



1.3. Best Expression Found

FITNESS: `99.9023`



2. Genetic Programming Design choices and Operators

2.1. Representation of Genotype

For my genotype, I used an Expression Tree, where each Node has a lambda function that takes in a float and outputs a float, and a list of its child expressions. The lambda function would make use of its child expression's functions somehow.

The simplest case is a Constant Expression. If the constant is k, then its lambda expression would just be `(float x) => k`, where this means given the input x it will output k. As you can see, the lambda does not depend on the variable k at all (which is why it is a constant). This formulation allows for all the flexibility one would need to represent any expression. For example, if I want to represent `sin(x)`, then I would have at the root node a Sin Expression Node, which has a lambda function of `(float x) -> sin(child[0]->lambda(x))`, and the Expression Node would have exactly one subexpression. If its child is a Constant Expression Node of value k, then `(float x) -> sin(child[0]->lambda(x))` would evaluate to `(float x) -> sin(k)`, which makes sense.

Similarly, Plus, Minus, Divide, and Multiply operators are defined to be Expressions with exactly two child expressions, and their lambdas are `(float x) => op(child[0]->lambda(x),child[1]->lambda(x))`, where op is the corresponding `+`,`-`,`/` and `*`.

If I want to evaluate an expression at x, I can do an in order tree traversal of the lambdas, which would give me the lambda that represents the entire expression tree.

2.2. Diversity

My first attempt at diversity was using Phenotypic diversity, which is a sample of the Absolute Mean Error between two expressions, but this did not prove to be very effective, not only because it took way too long to calculate, but also because it didn't motivate genotypic diversity which is useful for crossing over.

I ended up using a simple genotypic diversity metric which is the sum of the differences of all types of operators between two expressions. So `sin(x)` and `cos(x)` has a distance of 2 from each other, because the first expression has 1 sin node while the second has none (1 - 0), and the second has 1 cos node while the first has none (0 - 1), and both has a variable (1 - 1). Therefore, their distance is |1-0| + |0-1| + |1-1| = 2. This is more inline with the diversity I wanted to encourage, and I got better results with this.

With more time, I would like to experiment with structural diversity, which somehow takes into account the topology of the expression tree, because I think that might also be very useful for crossing over.

2.3. Probabilistic Deterministic Crowding

I invented a Reproduction Operator I call Probabilistic Deterministic Crowding. After it reproduces two parents through crossover and mutates the offspring a bit, half of the time, it puts both parents and the offspring back into the population, the other half, my reproduction operator is like deterministic crowding and the child replaces the most similar parents.

When both parents and the offspring is put back into the population, the individual with the worst fitness is removed from the population. I hypothesized that the method would able to maintain diversity (child replaces most similar parent), but still ensures a minimum level of improvement in the population's fitness.

Unfortunately, this method did not perform as well as I expected. It was slow because some the weakest individuals were unlikely to be removed from the population, yet the diversity still decayed too quickly with time. In other words, it inherited the worst properties of both methods.

I ended up just keeping the parent in the population and adding the offspring to the population. Then, when the population size is too large, I remove individuals from the bottom.

2.4. Solver

- `GenerationalSolver`: - The algorithm for this solver can be described as: 1. Use a selector to choose M parents 2. Create N (population size) offsprings from the parents 3. Replace the current population with the offspring 4. Repeat - This was what I used for my first assignment, but this did not give me very good results. I think this is because it removes too many good solutions from population in step 3. Further, it does not follow the motto of GP: "incremental progress".
- `ContinuousSolver`: - The algorithm for this solver can be described as: 1. Use a selector to choose 2 parents 2. Create 1 offspring from 2 parents 3. Decide which of the three individuals to keep and put back to the population. 4. Repeat - Not only does this solver resolves the two disadvantages of the previous solver, it also allows for the flexibility of a protocol for choosing how to handle choosing who to keep from the parents and the offspring. - Indeed, as expected, the results from this solver is better than the `GenerationalSolver`, so you will only see me discuss results from the `ContinuousSolver`.

2.5. Selectors

- `Tournament Selector`: This is the standard tournament selector in literature where N individuals are selected at random from the population, and the individual with the highest fitness gets to reproduce. This happens twice to give the two parents that would reproduce.
- `DiversitySelector`: This selector uses the `TournamentSelector` to "suggest" two individuals with high fitness, and this selector moves forward with the suggested parents only if the parents has a distance value larger than some value K. If this condition is not meet, K is decayed, then the process is repeated. This operator is named this way because it trys to select parents that are as different from each other as stochastically possible.
- `NichingSelector`: This selector is the opposite of `DiversitySelector`, moving forward with `TournamentSelector`'s suggestion only if the parents' distance value is less than a certain K. If not, K is incremented, then the process is repeated. This Operator aims to create various niches within the population.

2.6. Mutators

- `ConstantMultiplier`: This mutator came out of a conversation I had with Joni the TA about my program's performance. He noted that my expressions did not have a global constant multiplier term. This mutator

aims to resolve this, because when it is called to mutate an expression, it multiplies the expression by a constant and keeps the resulting expression if it has a higher fitness.
- `TrigMultiplier`: This mutator is similar to `ConstantMutator`, but it multiplies a trig function rather than a constant.
- `ConstantMutator`: finds a constant and mutates its k value
- `SubexpressionMutator`: Finds a subexpression tree, and swaps it out with a random expression tree. This is really only useful when the subexpression tree is not too large compared to the size of the full tree.
- `TruncateMutator`: Replaces a subexpression tree with a constant of random value

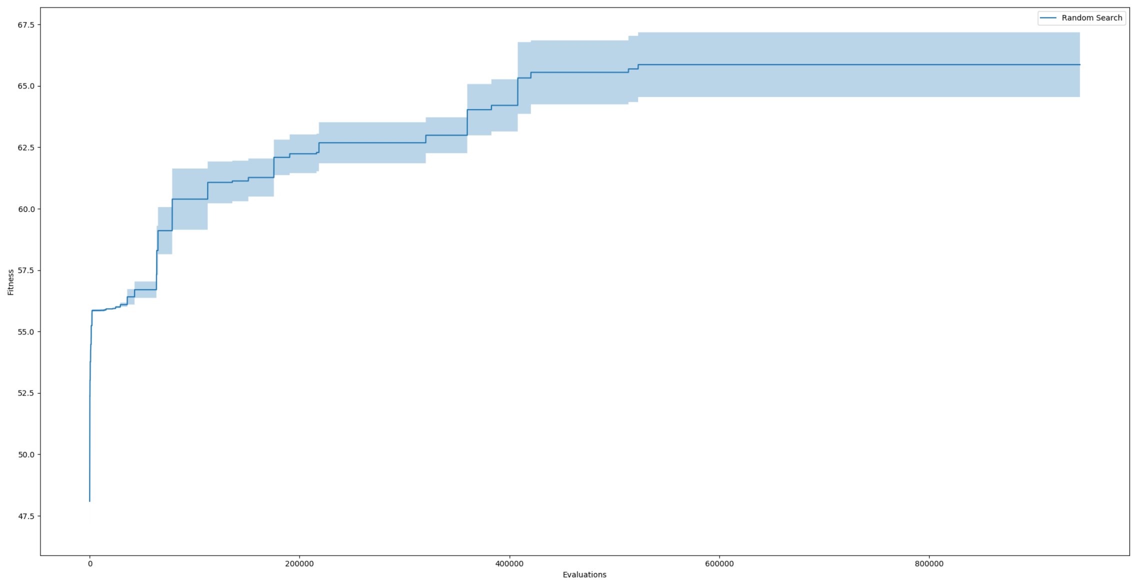
3. Random Search

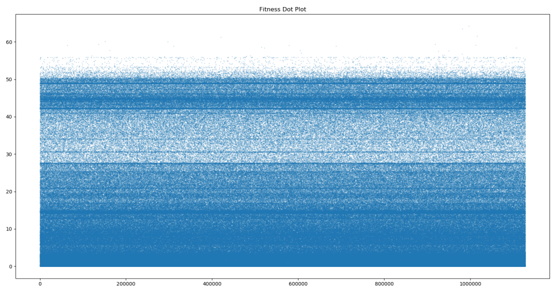
3.1 Description

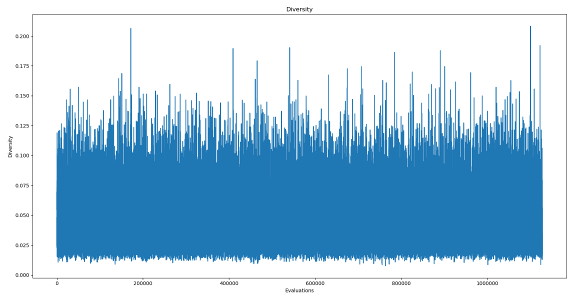
Random Search is a special case of a Genetic Programming with the only constraint that the Reproducer operator would output a random expression for any parents. Note that this puts no constraint on the Selection operator used (it can be anything, but ideally it would take the least amount of compute power possible, as the parent's fitness has no correlation with the child's fitness). Therefore, I just implemented my random search as using the `RandomReproducer`, which outputs a random expression for any two parents.

3.2 Plots

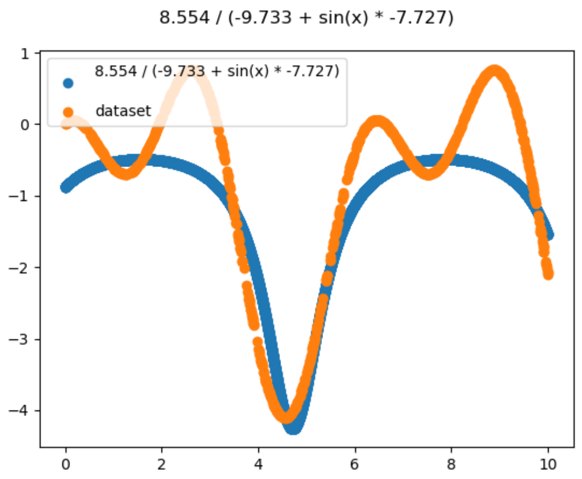
Learning Plot (Top), Dot Plot (Bottom Left), Diversity Plot (Bottom Right)

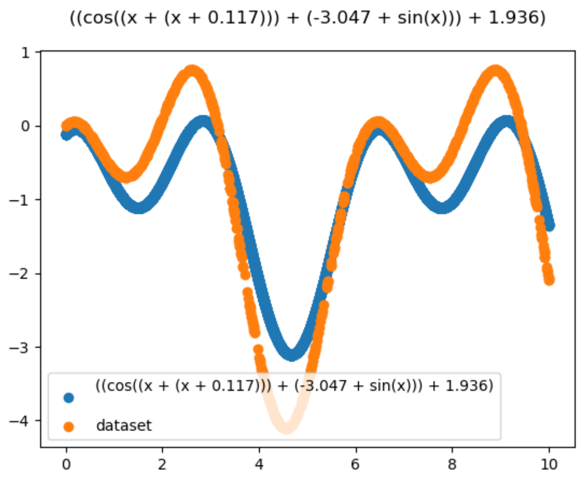


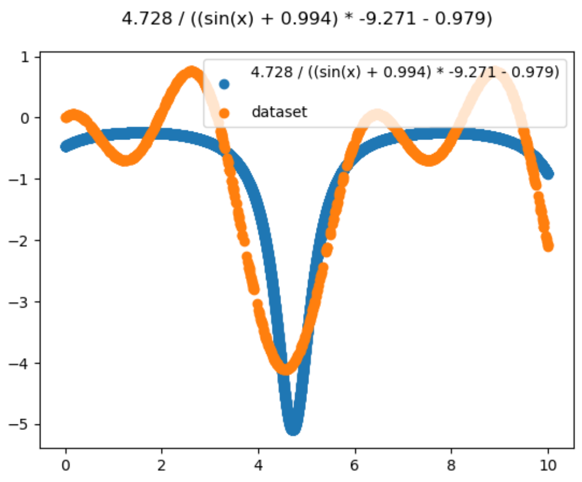


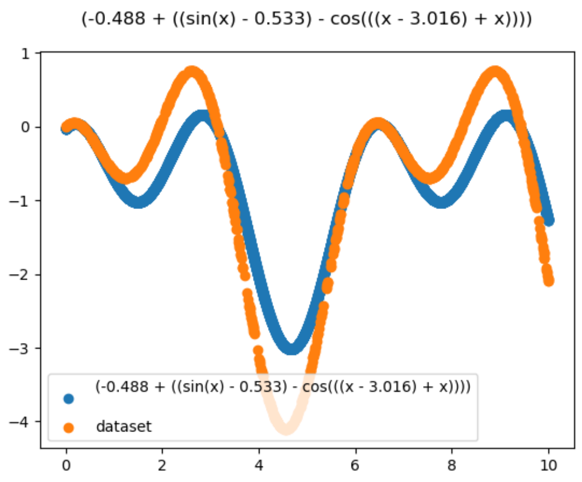


As you would expect, the dot plot shows an almost uniform distribution of fitnesses, which suggests great diversity. This is confirmed by the diversity plot. The learning curve is abrupt, as expected. What you would not expect is for a random search to do so well. Most of the final best equations were able to capture the shapes of the dataset quite well.









4. Hill Climber

4.1. Description

Similar to Random Search, Hill Climbers (HC) are just a special case of Evoluationary Algorithms, with population 1, simulated annealing with initial temperature set to 0, and a 100% mutation rate. In this case, it doesn't matter if the solver is generational or continuous, because only one individual is reproducing, and the individual is reproducing asexually.

What is important when talking about hill climbers is the AsexualReproducer. Unlike other reproducers, this reproducer takes one parent, mutates it until either the offspring has a fitness than the parent, or, by simulated annealing, it is let through. Of course, in HC, temperature is 0 so the child is only let through if it has a higher fitness than the parent.

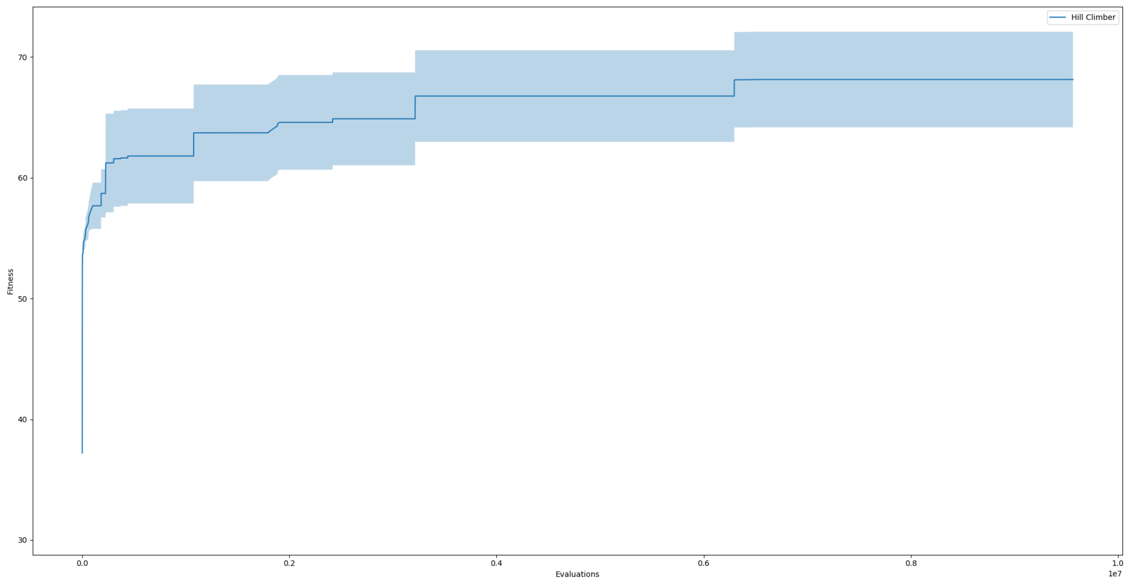
4.2. Plots

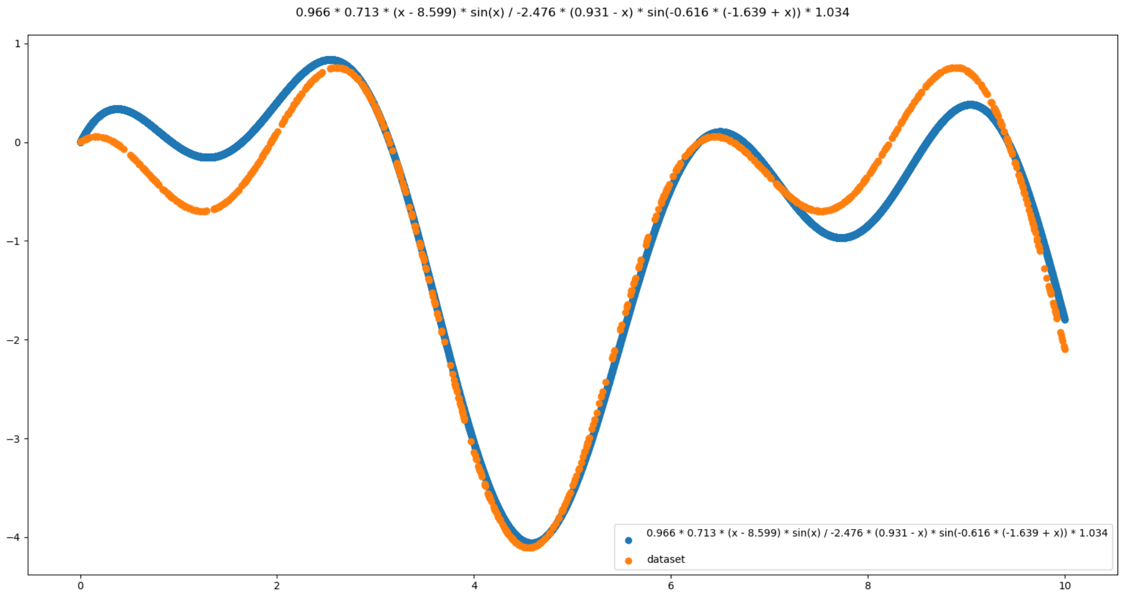
Hyperparameters

Below are the hyperparameters I used for Hill Climber

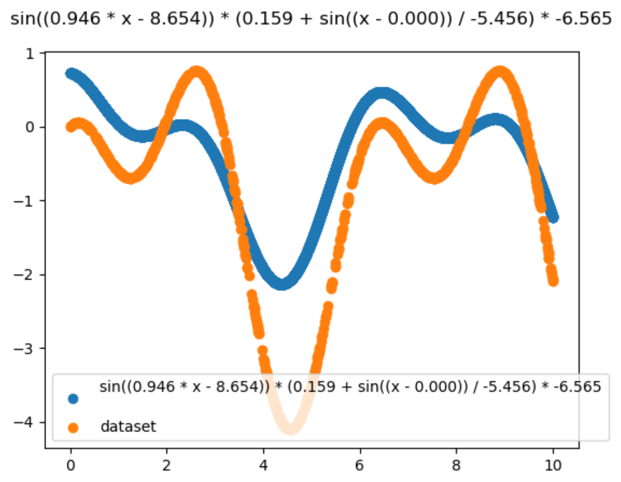
```
Input data.txt
PopulationCount 1
GenerationCount 100000
MaxDepth 15
MinThreads 5
MaxThreads 100
Reproducer Asexual
Solver Continuous
MutationRetries 50
Init_T 0
T_decay 1
```

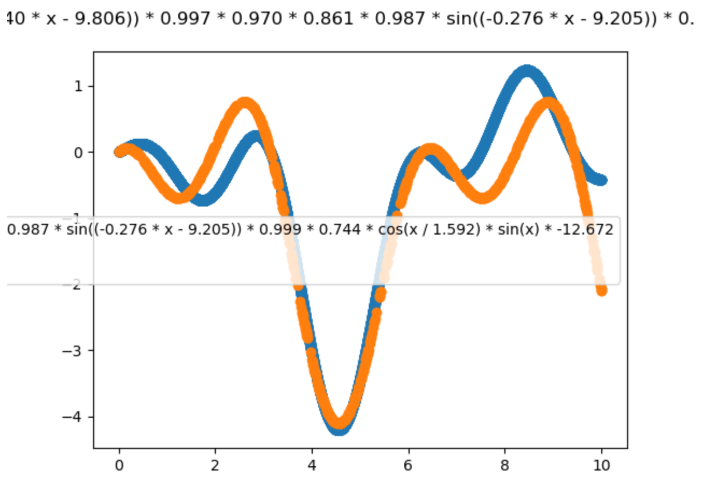
Learning Plot

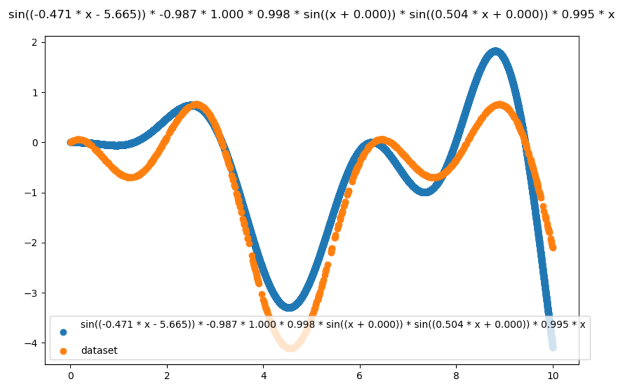


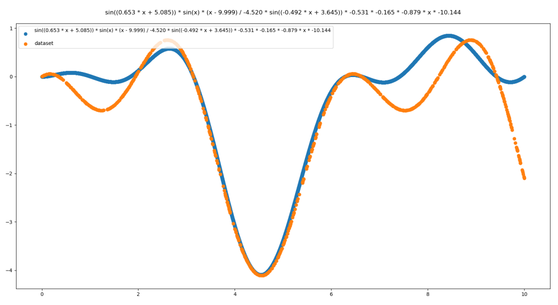


The equation above has a fitness of `82.646`, which is the highest fitness I got for HC. Overall, it does seem like HC performed better than Random Search.









5. Genetic Programming (GP)

5.1. Description

I experimented with various configurations of GP:

- `Base GP`: This GP uses the `DiversitySelector` (described above) with the continuous to select two parents (which in turn uses `TournamentSelector`). The expression tree of the parents are crossed over at a random point, which gives the base of the child. Then the child has some probability of mutating through all of the mutators described above. The child is then inserted into the population, then the individual with the lowest fitness is removed, and the cycle repeats. A population size of 50 is used.
- `Tournament GP`: similar to `Base GP`, but uses vanilla `TournamentSelector`.
- `Large Population GP`: as the name implies, this configurations tries to see if a large population of 500 would outperform a smaller population. I expect that it has a slower convergence, but would reach a better solution than smaller populations.

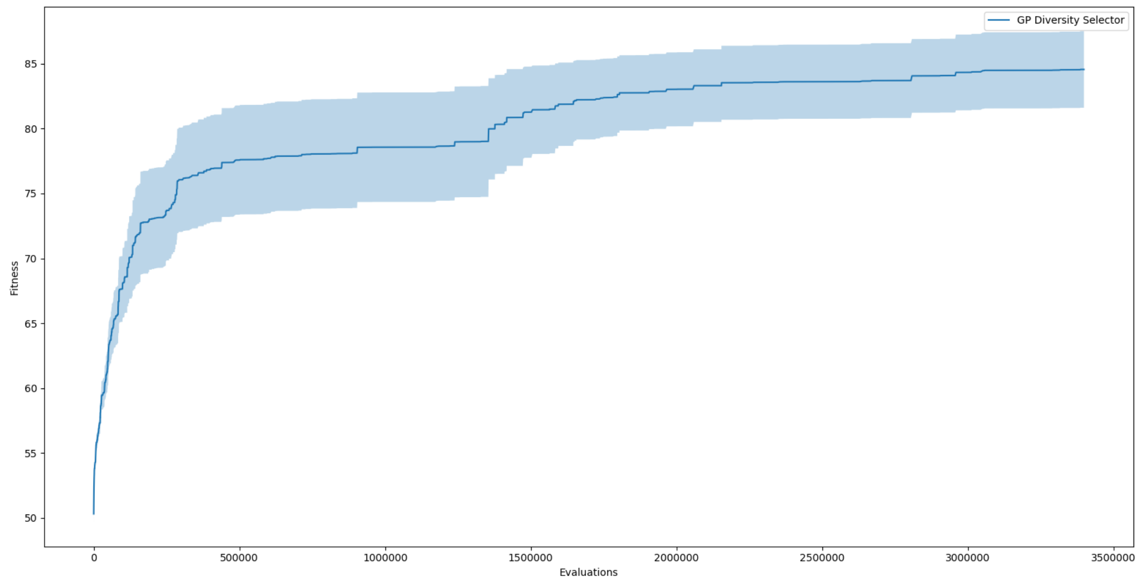
5.2. Base GP

5.2.1. Hyperparameters

Below are the hyperparameters I used for the Base GP

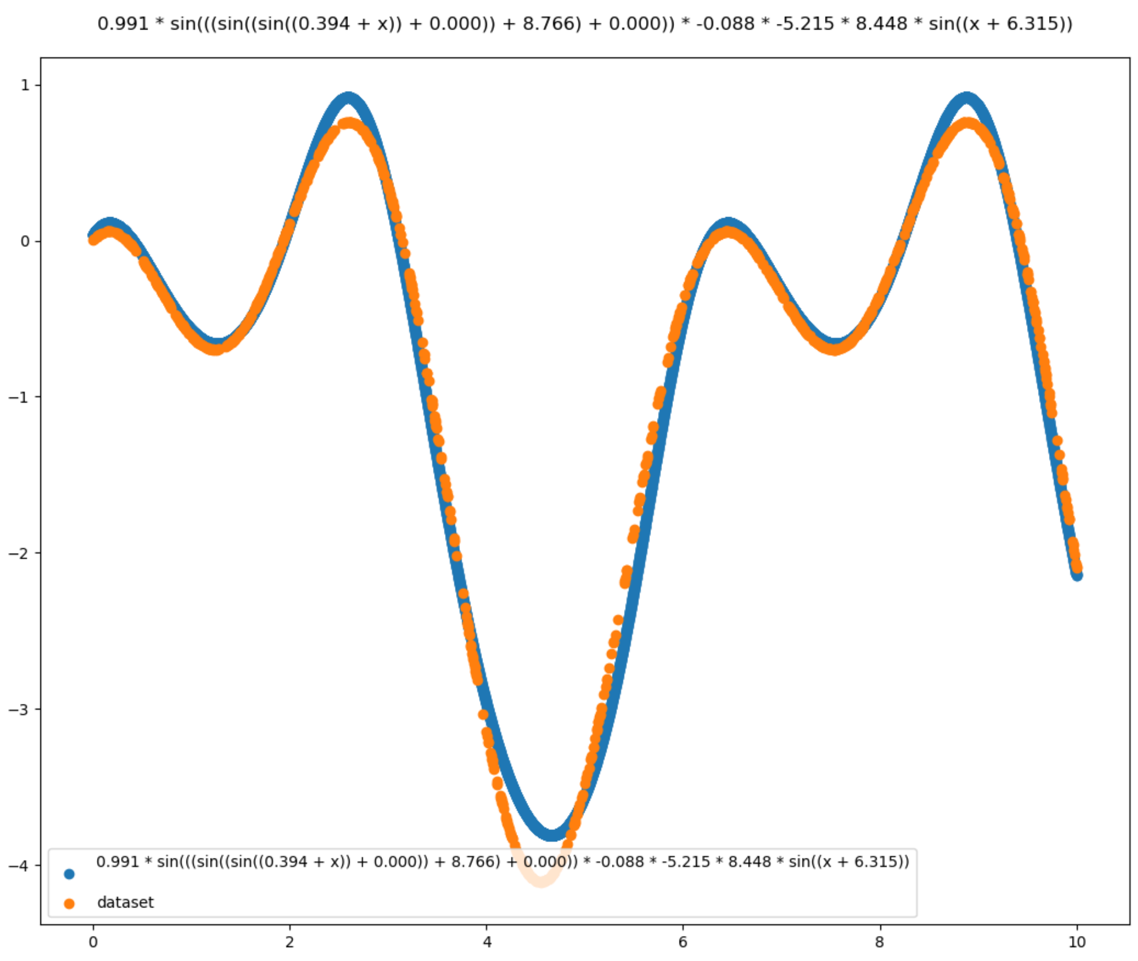
```
Input data.txt
Solver Continuous
Reproducer CrossoverMutator
Selector Diversity
PopulationCount 50
TournamentPlayersCount 4
MutationRetries 8
MaxDepth 10
MinThreads 5
MaxThreads 100
Init_T 0.5
T_decay 0.999
```

5.2.2. Learning Curve for Base GP

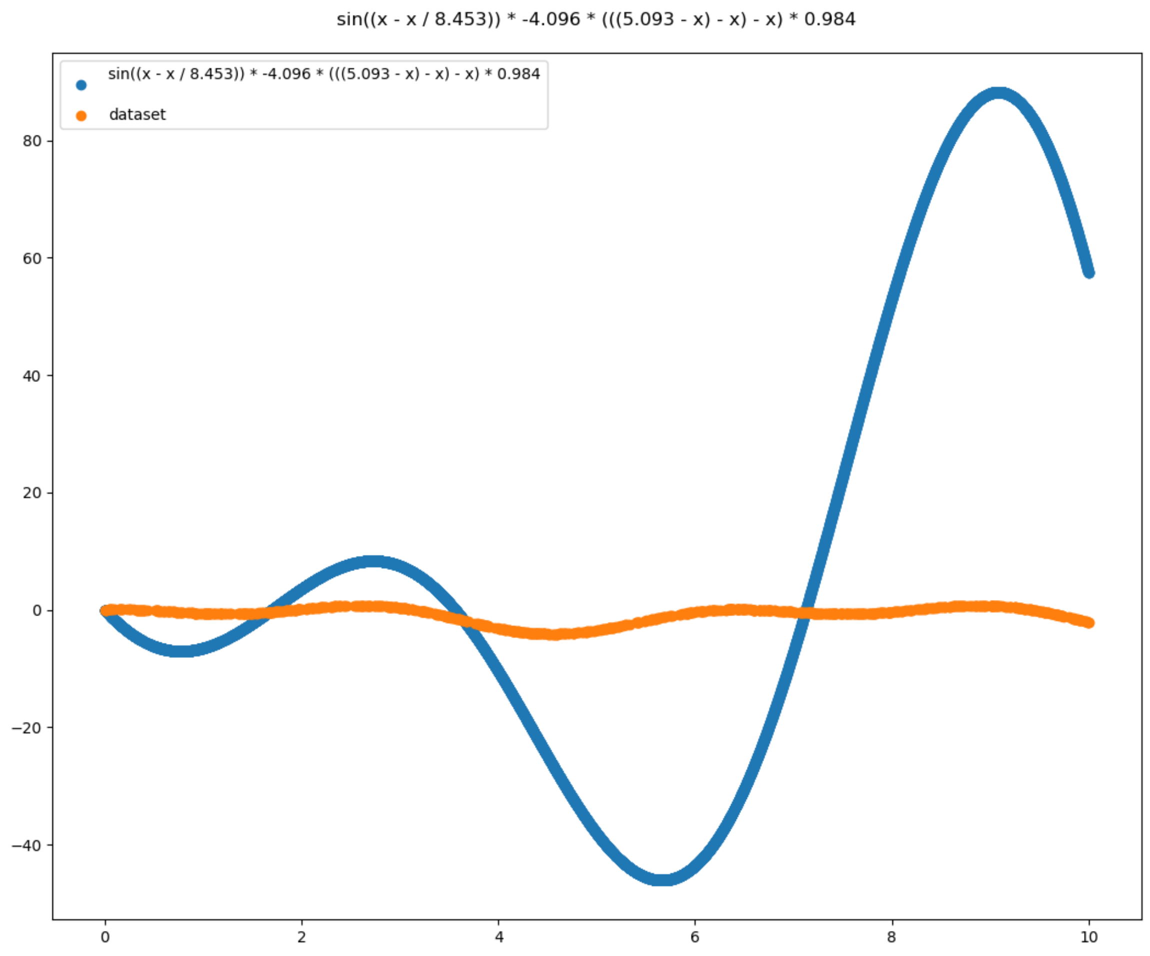


5.2.3. Expressions of Base GP

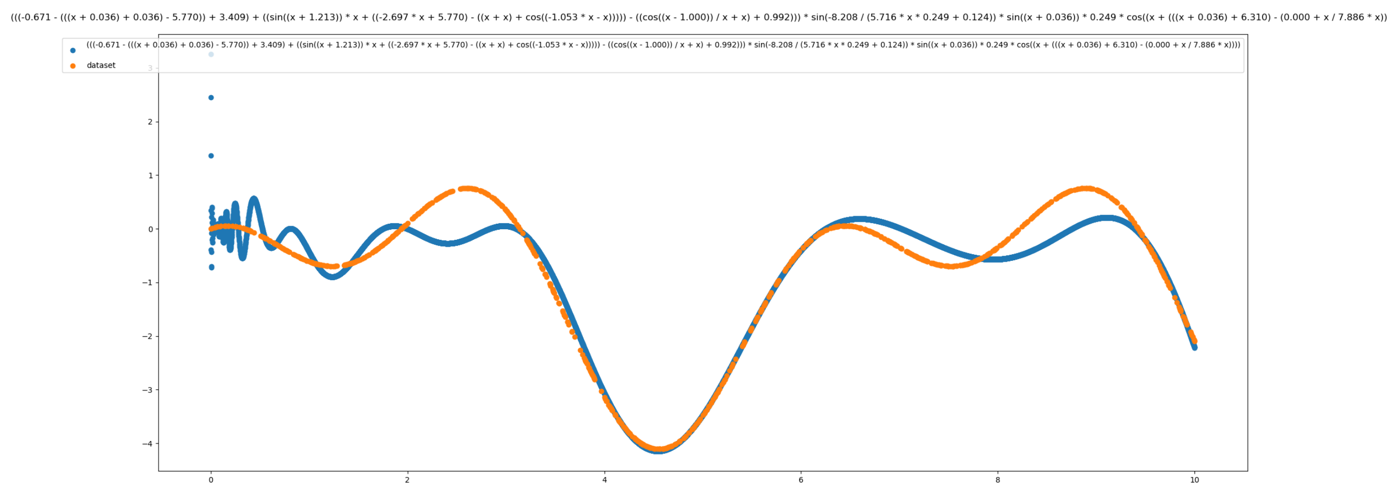
Base GP did not perform as well as I had expected. Though sometimes it got something pretty good ...

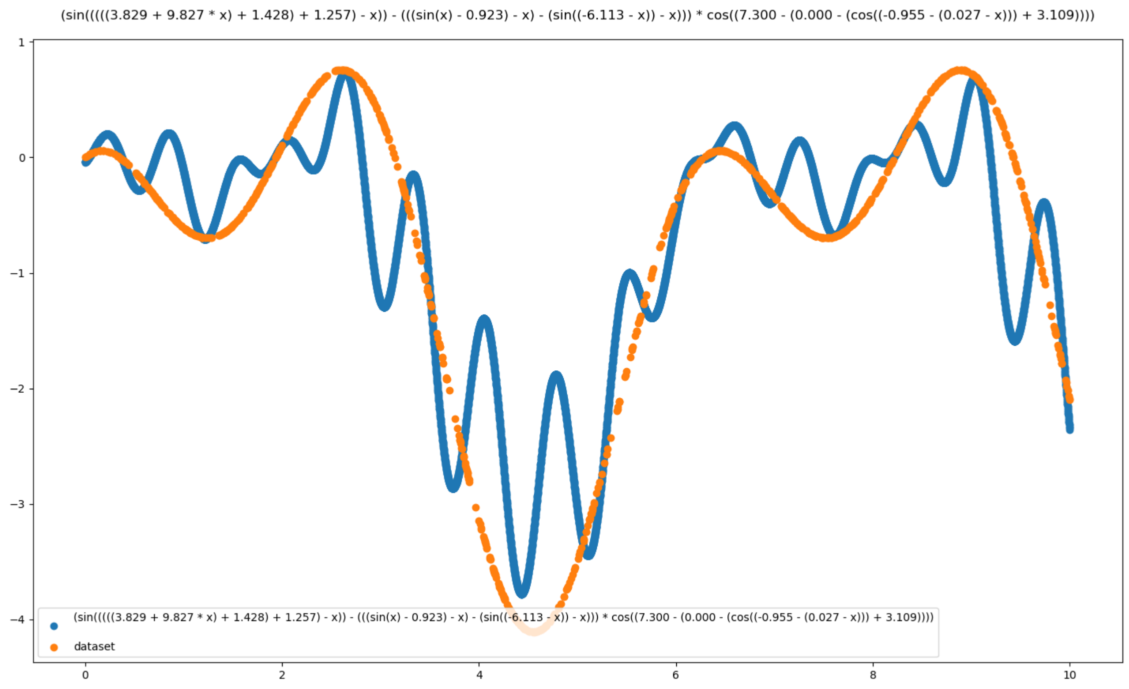


Some times it was way off...



Other times it went too crazy with the waves...





5.3. Tournament GP

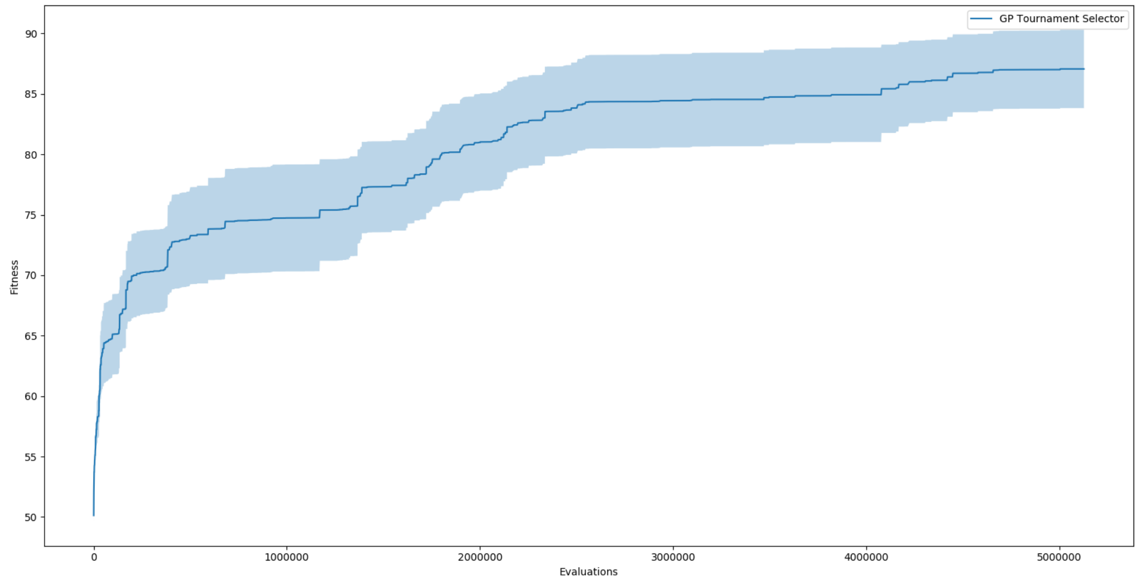
5.3.1. Hyperparameters

Below are the hyperparameters I used for the Tournament GP

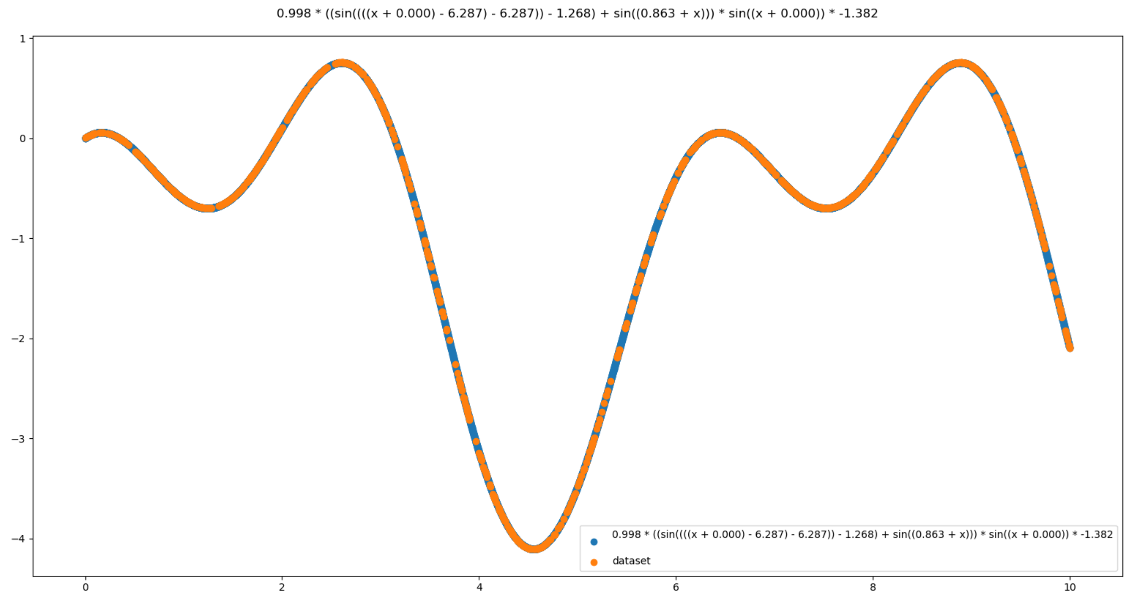
```
Input data.txt
Solver Continuous
Reproducer CrossoverMutator
Selector Tournament
PopulationCount 50
```

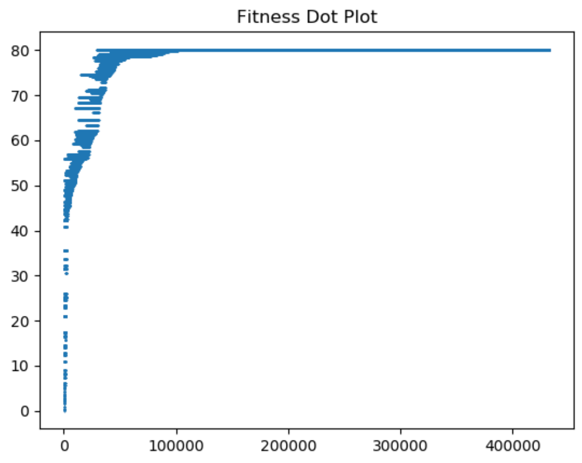
```
TournamentPlayersCount 4
MutationRetries 8
MaxDepth 10
MinThreads 5
MaxThreads 100
Init_T 0.5
T_decay 0.999
```

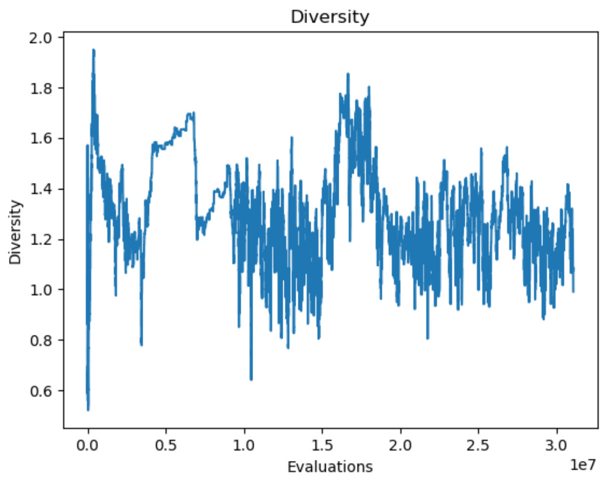
5.3.2. Learning Curve for Tournament GP



5.3.3. Expressions Generated by Tournament GP

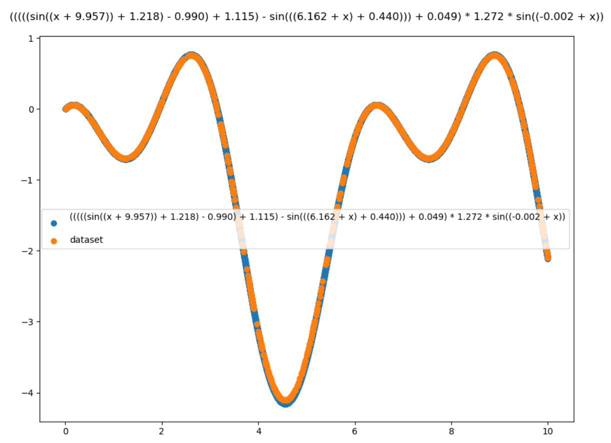


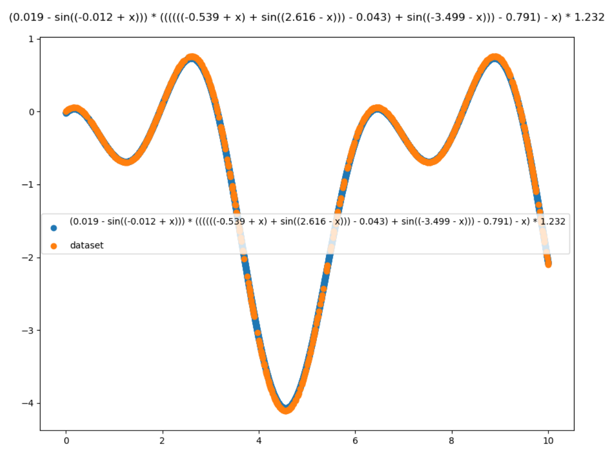


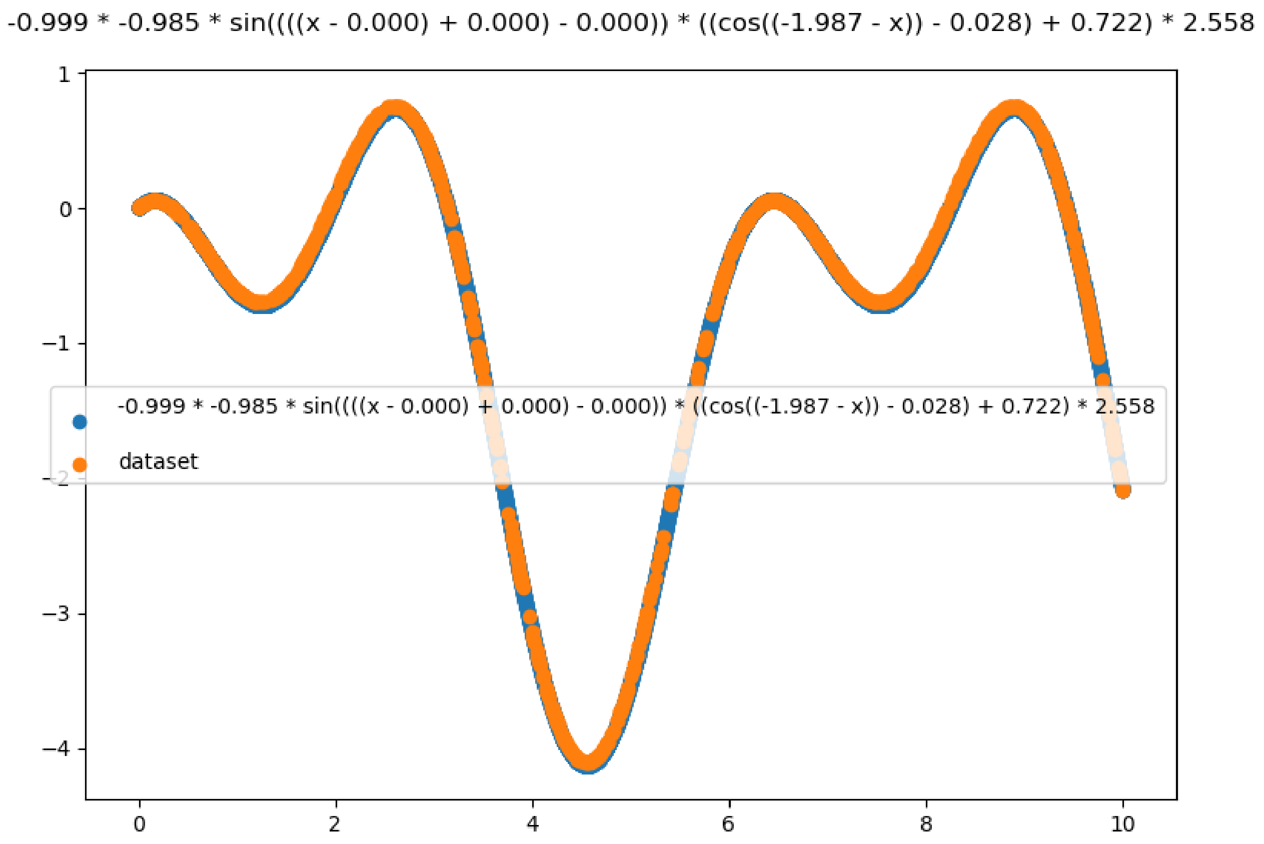


Above is the graph plot (top), dot plot (bottom left), and diversity plot (bottom right) of the Tournament GP that gave me the best expression. The Diversity plot clearly illustrates the effectiveness of the diversity, while the fitness dot plot shows that my population was able to find a diverse set of solutions that are all quite good. The best one, given by the equation:

Below are some graphs of other equations found by `Tournament GP`:







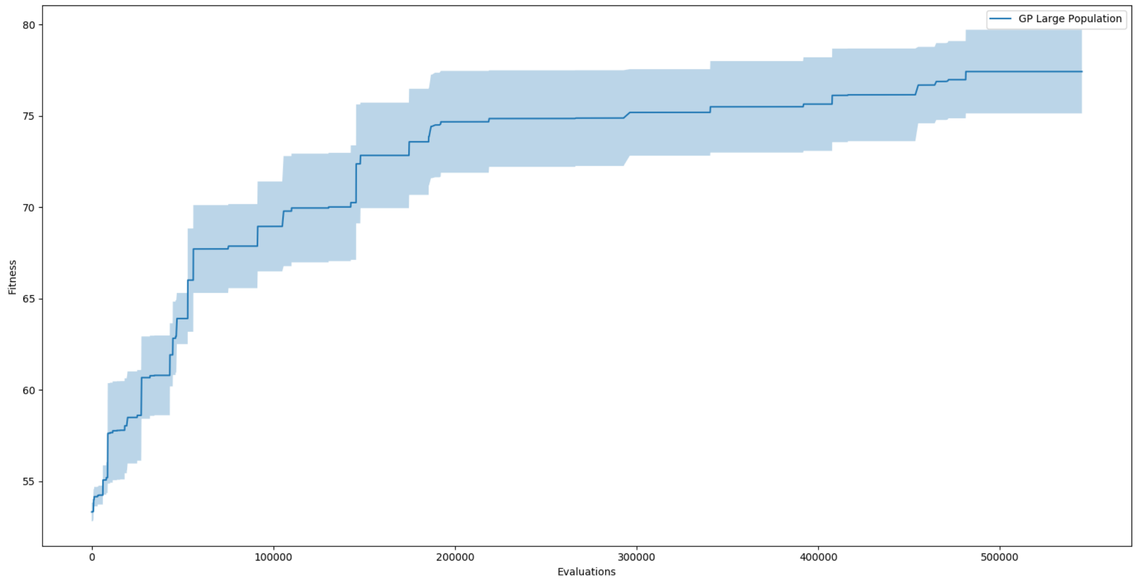
5.4. Large Population GP

Below are the hyperparameters I used for the Large Population GP

```
Input data.txt
Solver Continuous
Reproducer CrossoverMutator
Selector Diversity
PopulationCount 500
```

```
TournamentPlayersCount 4
MutationRetries 8
MaxDepth 10
MinThreads 5
MaxThreads 100
Init_T 0.5
T_decay 0.999
```

5.4.2. Learning Curve for Large Population GP



Because this run took so long, I was not able to let it finish, so I feel it would be unfair to include some expressions generated by this algorithm. However, from the learning curve, this method proves to be quite promising.

6. Analysis of Methods

6.1. Fitness Metric

The fitness of an individual in my program is given by `100 / (AbsoluteMeanError + 1)`. This metric is nice because it's almost like a progress bar to see how far it is away from have 0 error. When the error approaches infinity, this fitness approaches zero, and when the error approaches 0, the fitness approaches 100. This is also useful for the convergence plot.

NOTE: one of the requirements on the prompt was to show the `Overall efficiency of the algorithm (accuracy versus number of evaluations)`. This fitness metric is valid as a measure of accuracy, as accuracy of the expression increases with time, inversely proportional to the error. Therefore, all of the learning curves that have been presented in this document satisfies this requirement.

6.2. Simpler problems for testing

I have a segment of code in `Config.cpp` that would allow me to construct my own dataset rather than use input dataset, which means I'm able to create datasets of `sin(x)`,`cos(x)` and `x`. Usually, simple equations like this arise directly as a result of initialization, so it wasn't too helpful to study how my program would evolve these equations.

6.3. Conclusion

Overall, my `Tournament GP` was able to find the highest fitness of `99.90`. I have read in literature that Tournament Selector is usually a good go to selector for most problems because the selection pressure can easily be changed by changing the tournament size, and it has nice convergence as well as exploration properties. My studies have confirmed this. Not only was this GP able to find the best solution, but it consistently found very good solutions, compared to other methods I tried.

I expected HillClimber to do worse than GP simply because the space of all mathematical expressions that can be represented by an expression tree of max depth 10 is very large, so it loses out on not only the presence of other individuals in the population to crossover with, but also of exploring slightly worse solutions to get to better ones later on.

As mentioned above, Random Search performed much better than I thought. I think this is because though the search space is large, most of the inflections of the dataset can be captured by a small composition of some simple components, which explains why RandomSearch was able to reach such a high fitness instantly (as can be seen from the learning curve), then is able to push to `69.80`.

Appendix

```
#include <iostream>
#include <time.h>
#include "engine/Config.hpp"
#include <algorithm>
#include "genetic-programming/Solver.hpp"
#include "engine/OutputLogger.hpp"
#include <chrono>

#include "expression/Sin.hpp"
#include "expression/Cos.hpp"
#include "expression/Variable.hpp"

using namespace std;
using namespace std::chrono;
using namespace SymbolicRegression;
int main(int argc, char **argv)
{
    int seed = int(time(NULL));
    string configFile = "";
    string outputDir = "tmp";
    if (argc > 1)
    {
        outputDir = string(argv[1]);
    }
    if (argc > 2)
    {
        // output dir and config supplied
        configFile = string(argv[2]);
    }
    if (argc > 3)
    {
        seed = stoi(argv[3]);
    }
    cout << endl
         << "SEED:" << seed << endl
         << endl;
    srand(seed);
    try
    {
        Config config(configFile + ".config", outputDir);
        OutputLogger::Log("Config", "Seed:" + to_string(seed));
        OutputLogger::Log("Config", string(config));
        cout << string(config) << endl;
        auto solver = Solver::Instance();
        auto start = high_resolution_clock::now();
        solver->Run();
        auto stop = high_resolution_clock::now();
        auto ms = duration_cast<milliseconds>(stop - start);
        float seconds = ms.count() / 1000.0f;
        cout << "Total Time of Run: " << seconds << endl;
    }
    catch (exception e)
    {
        cout << e.what() << endl;
        return -1;
    }
}
```

Mutators

ConstantMultiplierMutator.hpp

```
#ifndef _CONSTANT_MULTIPLIER_MUTATOR_HPP_
#define _CONSTANT_MULTIPLIER_MUTATOR_HPP_
#include <memory>
#include "../../expression/Expression.hpp"
namespace SymbolicRegression
{
using namespace std;
class ConstantMultiplierMutator
{
public:
    static shared_ptr<Expression> Mutate(const shared_ptr<Expression> &exp);
};
} // namespace SymbolicRegression

#endif
```

ConstantMultiplierMutator.cpp

```
#include "ConstantMultiplierMutator.hpp"
#include <iostream>
#include <vector>
#include <algorithm>
#include "../../expression/Constant.hpp"
#include "../../expression/Multiply.hpp"
#include "../../engine/Config.hpp"

namespace SymbolicRegression
{
using namespace std;

// TRY OUT SAME EXPRESSION WITH A DIFFERENT CONSTANT MULTIPLIER
shared_ptr<Expression> ConstantMultiplierMutator::Mutate(const
shared_ptr<Expression> &exp)
{
    float k, prevFitness;
    auto tempExp = Expression::Copy(exp);
    prevFitness = tempExp->Fitness();
    vector<shared_ptr<Expression>>::iterator it;
    for (int i = 0; i < Config::GetInt("MutationRetries"); i++)
    {
        // Find subexpression of type constant
        it = find_if(tempExp->m_subexpressions.begin(),
                     tempExp->m_subexpressions.end(),
                     [](auto subexp) {
                         return EXPRESSION_TYPE(subexp) == CONSTANT_T;
                     });
        // Tree is already too tall, and don't already
        // have constants multiplied in top level
        // to be mutated
        if (tempExp->Depth() >= Config::GetInt("MaxDepth") &&
            it == tempExp->m_subexpressions.end())
        {
            break;
        }

        k = Expression::RandomF(-2, 2);
        auto f = [=](float x) { return k; };
        if (tempExp->Fitness(f) > prevFitness)
        {
            // If expression already has multiply expression in top level
            // and constant in second level
            if (EXPRESSION_TYPE(tempExp) == MULTIPLY_T &&
                it != tempExp->m_subexpressions.end())
            {
                auto constExp = shared_ptr<Expression>(new Constant(1, k));
                replace(tempExp->m_subexpressions.begin(),
                        tempExp->m_subexpressions.end(),
                        *it, constExp);
                prevFitness = tempExp->Fitness();
            }
            // Create new constant and multiplier expression
            else
            {
                auto constExp = shared_ptr<Expression>(new Constant(1, k));
                tempExp = shared_ptr<Expression>(new Multiply(0, constExp,
tempExp));
                prevFitness = tempExp->Fitness();
            }
        }
    }

    return exp;
}
} // namespace SymbolicRegression
```

ConstantMutator.hpp

```
#ifndef _CONSTANT_MUTATOR_HPP_
#define _CONSTANT_MUTATOR_HPP_
```

```
#include <memory>
#include "../../expression/Expression.hpp"
namespace SymbolicRegression
{
using namespace std;
class ConstantMutator
{
public:
    static shared_ptr<Expression> Mutate(const shared_ptr<Expression> &exp);
};
} // namespace SymbolicRegression

#endif
```

ConstantMutator.cpp

```
#include "ConstantMutator.hpp"
#include <iostream>
#include <vector>
#include <algorithm>
#include "../../expression/Constant.hpp"
#include "../Solver.hpp"
#include "../../engine/Config.hpp"

namespace SymbolicRegression
{
using namespace std;
shared_ptr<Expression> ConstantMutator::Mutate(const shared_ptr<Expression> &exp)
{
    auto tempExp = Expression::Copy(exp);
    auto collapsedExp = tempExp->Collapse(tempExp);
    vector<shared_ptr<Expression>> constants;
    int i = int(Expression::RandomF(0, 100));
    copy_if(
        collapsedExp->begin(),
        collapsedExp->end(),
        back_inserter(constants),
        [](auto e) {
            return EXPRESSION_TYPE(e) == CONSTANT_T;
        });
    if (constants.size() == 0)
        return exp;
    i = i % constants.size();
    auto constantToMutate = dynamic_pointer_cast<Constant>(constants[i]);

    float k, bestK;
    float prevFitness = tempExp->Fitness();
    float testFitness = -1;
    for (int i = 0; i < Config::GetInt("MutationRetries"); i++)
    {
        auto f = [=](float x) { return k; };
        testFitness = tempExp->Fitness(f);
        if (testFitness > prevFitness || Expression::RandomF() < Solver::GetTemp())
        {
            prevFitness = testFitness;
            bestK = k;
        }
    }

    if (prevFitness > exp->Fitness() || Expression::RandomF() < Solver::GetTemp())
    {
        constantToMutate->m_k = bestK;
        return tempExp;
    }
    return exp;
}
} // namespace SymbolicRegression
```

SubexpressionMutator.hpp

```
#ifndef _SUBEXPRESSION_MUTATOR_HPP_
#define _SUBEXPRESSION_MUTATOR_HPP_
#include <memory>
#include "../../expression/Expression.hpp"
namespace SymbolicRegression
{
using namespace std;
class SubexpressionMutator
{
public:
    static shared_ptr<Expression> Mutate(shared_ptr<Expression> exp);
};
} // namespace SymbolicRegression

#endif
```

SubexpressionMutator.cpp

```
#include "SubexpressionMutator.hpp"
#include <vector>
#include <algorithm>
#include "../../engine/Config.hpp"
#include "../Solver.hpp"
```

```
namespace SymbolicRegression
{
using namespace std;

shared_ptr<Expression> SubexpressionMutator::Mutate(shared_ptr<Expression> exp)
{
    shared_ptr<Expression> bestMutation;
    for (int i = 0; i < Config::GetInt("MutationRetries"); i++)
    {
        auto tempExp = Expression::Copy(exp);
        auto collapsedExp = exp->Collapse(exp);
        vector<shared_ptr<Expression>> operators;
        copy_if(
            collapsedExp->begin(),
            collapsedExp->end(),
            back_inserter(operators),
            [](auto e) {
                return e->Order() > 0;
            });
        if (operators.size() == 0)
            return exp;

        // choose random operator
        auto expToChange = operators[int(Expression::RandomF(0, 100)) %
                                     operators.size()];

        // change a random subexpression of the operator
        int idx = int(Expression::RandomF(0, float(expToChange->Order() - 1)));
        expToChange->m_subexpressions[idx] =
            Expression::GenerateRandomExpression(expToChange->Level() + 1);
        if (!bestMutation || tempExp->Fitness() > bestMutation->Fitness())
        {
            bestMutation = tempExp;
        }
    }

    if (bestMutation->Fitness() > exp->Fitness() ||
        Expression::RandomF() < Solver::GetTemp())
    {
        return bestMutation;
    }
    return exp;
}
} // namespace SymbolicRegression
```

TrigMultiplierMutator.hpp

```
#ifndef _TRIG_MULTIPLIER_MUTATOR_HPP_
#define _TRIG_MULTIPLIER_MUTATOR_HPP_
#include <memory>
#include "../../expression/Expression.hpp"
namespace SymbolicRegression
{
using namespace std;
class TrigMultiplierMutator
{
public:
    static shared_ptr<Expression> Mutate(const shared_ptr<Expression> &exp);
};
} // namespace SymbolicRegression

#endif
```

TrigMultiplierMutator.cpp

```
#include "TrigMultiplierMutator.hpp"
#include <iostream>
#include <vector>
#include <algorithm>
#include "../../expression/Constant.hpp"
#include "../../expression/Variable.hpp"
#include "../../expression/Multiply.hpp"
#include "../../expression/Plus.hpp"
#include "../../expression/Sin.hpp"
#include "../../engine/Config.hpp"

namespace SymbolicRegression
{
using namespace std;
shared_ptr<Expression> TrigMultiplierMutator::Mutate(const shared_ptr<Expression>
&exp)
{
    float frequency, displacement, output_frequency, output_displacement;
    float prevFitness, testFitness;
    prevFitness = exp->Fitness();
    for (int i = 0; i < Config::GetInt("MutationRetries"); i++)
    {
        if (exp->Depth() >= Config::GetInt("MaxDepth"))
        {
            break;
        }
        frequency = Expression::RandomF(-10, 10);
        displacement = Expression::RandomF(-10, 10);
        auto f = [=](float x) {
            return sinf(frequency * x + displacement);
        };
        testFitness = exp->Fitness(f);
        if (testFitness > prevFitness)
        {
            prevFitness = testFitness;
            output_frequency = frequency;
            output_displacement = displacement;
        }
    }

    if (prevFitness == exp->Fitness())
    {
        return exp;
    }
    auto frequencyConstant = shared_ptr<Expression>(new Constant(-1,
output_frequency));
    auto displacementConstant = shared_ptr<Expression>(new Constant(-1,
output_displacement));
    auto variable = shared_ptr<Expression>(new Variable(-1));

    auto variableTimesFrequency = shared_ptr<Expression>(new Multiply(0,
frequencyConstant, variable));
    auto signalTerm = shared_ptr<Expression>(new Plus(0, variableTimesFrequency,
displacementConstant));

    auto sinExpression = shared_ptr<Expression>(new Sin(0, signalTerm));
    return shared_ptr<Expression>(new Multiply(0, sinExpression, exp));
}
} // namespace SymbolicRegression
```

TruncateMutator.hpp

```
#ifndef _TRUNCATE_MUTATOR_HPP_
#define _TRUNCATE_MUTATOR_HPP_
#include <memory>
#include "../../expression/Expression.hpp"
namespace SymbolicRegression
{
using namespace std;
class TruncateMutator
{
public:
    static shared_ptr<Expression> Mutate(const shared_ptr<Expression> &exp);
};
} // namespace SymbolicRegression
```

```
#endif
```

TruncateMutator.cpp

```
#include "TruncateMutator.hpp"
#include <iostream>
#include <vector>
#include <algorithm>
#include "../../expression/Constant.hpp"
#include "../Solver.hpp"
#include <typeinfo>
#include "../../engine/Config.hpp"
namespace SymbolicRegression
{
using namespace std;
shared_ptr<Expression> TruncateMutator::Mutate(const shared_ptr<Expression> &exp)
{
    shared_ptr<Expression> bestMutation;
    for (int i = 0; i < Config::GetInt("MutationRetries"); i++)
    {
        auto tempExp = Expression::Copy(exp);
        auto collapsedExp = tempExp->Collapse(tempExp);
        vector<shared_ptr<Expression>> operators;
        copy_if(
            collapsedExp->begin(),
            collapsedExp->end(),
            back_inserter(operators),
            [](auto e) {
                return e->Order() > 0;
            });
        if (operators.size() == 0)
            return exp;
        auto opToTruncate = operators[int(Expression::RandomF(0, 100)) %
                                      operators.size()];
        opToTruncate->m_subexpressions[int(Expression::RandomF(0, 100)) %
                                       opToTruncate->m_subexpressions.size()] =
            Expression::GenerateRandomZeroOrderExpression(opToTruncate->Level() +
1);
        if (!bestMutation || tempExp->Fitness() > bestMutation->Fitness())
        {
            bestMutation = tempExp;
        }
    }

    if (bestMutation->Fitness() > exp->Fitness() ||
        Expression::RandomF() < Solver::GetTemp())
    {
        return bestMutation;
    }
    return exp;
}
} // namespace SymbolicRegression
```

Engine

Config.cpp

```
#include "Config.hpp"
#include <fstream>
#include <iostream>
#include <sstream>
#include <algorithm>
#include <iterator>
namespace SymbolicRegression
{
using namespace std;

Config *Config::Instance = nullptr;
shared_ptr<vector<tuple<float, float>>> Config::Data(new vector<tuple<float, float>>
());

Config::Config(string configFilePath, string outputFilePath)
{
    m_configs.insert(make_pair("Input", "data.txt"));

    m_configs.insert(make_pair("Solver", "Continuous"));
    m_configs.insert(make_pair("Reproducer", "CrossoverMutator"));
    m_configs.insert(make_pair("Selector", "Diversity"));
    m_configs.insert(make_pair("MinThreads", "20"));
    m_configs.insert(make_pair("MaxThreads", "-1"));
    m_configs.insert(make_pair("MaxDepth", "5"));

    m_configs.insert(make_pair("Init_T", "0.5"));
    m_configs.insert(make_pair("T_decay", "0.999999"));
    m_configs.insert(make_pair("MutationRetries", "1"));

    m_configs.insert(make_pair("PopulationCount", "100"));
    m_configs.insert(make_pair("GenerationCount", "50000"));
    m_configs.insert(make_pair("ElitesCount", "1"));
    m_configs.insert(make_pair("DotPlot", "1"));
    m_configs.insert(make_pair("OutputPath", "runs/" + outputFilePath + "/"));

    m_configs.insert(make_pair("TournamentPlayersCount", "4"));
    Instance = this;
    if (configFilePath != ".config")
    {
        string buf;
        ifstream configFile;
        configFile.open("configs/" + configFilePath);
        if (!configFile.is_open())
            BadConfigFile("Can't open file");
        while (getline(configFile, buf))
            ParseConfigLine(buf);
        configFile.close();
    }
    ParseInputDatapoints(m_configs["Input"]);
    // UseTestInputDatapoints();
}

void Config::UseTestInputDatapoints()
{
    auto f1 = [](float x) {
        return x;
    };
    auto f2 = [](float x) {
        return cosf(x) * 1.5f;
    };
    for (float i = 0; i < 10.0f; i += 0.01f)
    {
        Data->push_back(tuple<float, float>(i, f2(i)));
    }
}

void Config::ParseInputDatapoints(string inputFilePath)
{
    ifstream inputFile;
    inputFile.open("inputs/" + inputFilePath);
    if (!inputFile.is_open())
        BadConfigFile("Can't open file");
    vector<string> lines;
    copy(istream_iterator<string>(inputFile), istream_iterator<string>(),
back_inserter(lines));
    if (lines.size() % 2 != 0)
        BadConfigFile("Input Data in wrong format");
    for (int i = 0; i < lines.size() / 2; i++)
    {
        Data->push_back(tuple<float, float>(stof(lines[2 * i]), stof(lines[2 * i +
1])));
    }
}

int Config::GetInt(std::string key)
{
    if (IsValidKey(key))
    {
        return stoi(Instance->m_configs.at(key));
    }
    cout << "Invalid Key" + key << endl;
    throw std::exception(("Invalid Key" + key).c_str());
}
float Config::GetFloat(std::string key)
```

```
{
    if (IsValidKey(key))
    {
        return stof(Instance->m_configs.at(key));
    }
    cout << "Invalid Key" + key << endl;
    throw std::exception(("Invalid Key" + key).c_str());
}
std::string Config::GetString(std::string key)
{
    if (IsValidKey(key))
    {
        return Instance->m_configs.at(key);
    }
    cout << "Invalid Key" + key << endl;
    throw std::exception(("Invalid Key" + key).c_str());
}

Config::operator string() const
{
    string output = "";
    for (map<string, string>::const_iterator it = m_configs.begin(); it !=
m_configs.end(); ++it)
    {
        output += it->first + ":" + it->second + "\n";
    }
    return output;
}
void Config::ParseConfigLine(string configLine)
{

    vector<string> fields{
        istream_iterator<string>(stringstream(configLine)),
        istream_iterator<string>()};
    if (fields.size() != 2)
        BadConfigFile("Wrong format \"" + configLine + "\"");
    if (IsValidKey(fields[0]))
        m_configs[fields[0]] = fields[1];
    else
        BadConfigFile("Invalid Key \"" + fields[0] + "\"");
}

bool Config::IsValidKey(string key)
{
    return Instance->m_configs.find(key) != Instance->m_configs.end();
}

void Config::BadConfigFile(string message)
{
    throw exception(("Bad Config File: " + message).c_str());
}
} // namespace SymbolicRegression
```

Config.hpp

```
#ifndef _CONFIG_HPP_
#define _CONFIG_HPP_
#include <map>
#include <string>
#include <vector>
#include <tuple>
#include <memory>
namespace SymbolicRegression
{
using namespace std;
class Config
{
public:
    Config(string configFilePath, string outputFilePath);
    static Config *Instance; // Singleton
    static shared_ptr<vector<tuple<float, float>>> Data;

public:
    static int GetInt(string key);
    static float GetFloat(string key);
    static string GetString(string key);
    operator string() const;

private:
    void BadConfigFile(string message);
    void ParseConfigLine(string configLine);
    static bool IsValidKey(string key);
    void ParseInputDatapoints(string inputFilePath);
    void UseTestInputDatapoints();

protected:
    map<string, string> m_configs;
};
} // namespace SymbolicRegression
#endif
```

OutputLogger.cpp

```
#include "OutputLogger.hpp"
#include <memory>
```

```
#include <iostream>
#include "../genetic-programming/Solver.hpp"
#include <algorithm>
#include <cstdarg>
namespace SymbolicRegression
{
using namespace std;

int OutputLogger::Evaluations = 0;
shared_ptr<OutputLogger> OutputLogger::m_instance(new OutputLogger());
mutex OutputLogger::evalMu;

shared_ptr<OutputLogger> OutputLogger::Instance()
{
    return m_instance;
}

OutputLogger::OutputLogger()
{
    m_log.insert(pair<string, string>("Config", ""));
    m_log.insert(pair<string, string>("HighestFitness", ""));
    m_log.insert(pair<string, string>("FinalBest", ""));
    m_log.insert(pair<string, string>("FitnessDotPlot", ""));
    m_log.insert(pair<string, string>("Diversity", ""));
}

shared_ptr<vector<string>> OutputLogger::GetKeys()
{
    shared_ptr<vector<string>> keys(new vector<string>());
    keys->reserve(m_instance->m_log.size());
    for (auto const &it_log : m_instance->m_log)
        keys->push_back(it_log.first);
    return keys;
}

void OutputLogger::Clear(string key)
{
    m_instance->m_log[key] = "";
}

int OutputLogger::GetEvaluations()
{
    lock_guard<mutex> lock(evalMu);
    return Evaluations;
}

void OutputLogger::IncrementEvaluations()
{
    lock_guard<mutex> lock(evalMu);
    Evaluations = Evaluations + 1;
}

void OutputLogger::Log(string key, string log)
{
    m_instance->m_log[key] += log + "\n";
}

string OutputLogger::Get(string key)
{
    return m_instance->m_log[key];
}
} // namespace SymbolicRegression
```

OutputLogger.hpp

```
#ifndef _OUTPUT_LOGGER_HPP_
#define _OUTPUT_LOGGER_HPP_
#include <map>
#include <string>
#include <memory>
#include <mutex>
#include <vector>
namespace SymbolicRegression
{
using namespace std;
class OutputLogger
{
public:
    OutputLogger();
    static void Log(string key, string log);
    static string Get(string key);
    static shared_ptr<OutputLogger> Instance(); // Singleton
    static void IncrementEvaluations();
    static int GetEvaluations();
    static shared_ptr<vector<string>> GetKeys();
    static void Clear(string key);

private:
    map<string, string> m_log;
    static shared_ptr<OutputLogger> m_instance;
    static int Evaluations;
    static mutex evalMu;
};
} // namespace SymbolicRegression
#endif
```

Scripts

plotter.py

```
import matplotlib.pyplot as plt
import argparse
import numpy as np
import sys
import math
import os
import csv
import itertools


def import_finalbest(filepath):
    f = open(filepath, "r")
    lines = f.readlines()
    lines.reverse()
    eqn = lines.pop()
    print(eqn)
    points = [x.split(' ')
              for x in lines]
    x = []
    y = []
    for point in points:
        x.append(float(point[0]))
        y.append(float(point[1]))
    return x, y, filepath, eqn


def get_dir_name(runNumber):
    runs = [runs for runs in os.listdir(
        'runs') if os.path.isdir(os.path.join('runs', runs))]
    try:
        for run in runs:
            if run.find("run{}".format(runNumber)) is not -1:
                return "runs/{}/".format(run)
    except:
        print("Run {} not found!".format(runNumber))
        exit()


def read_csv(filepath, d=','):
    results = []
    with open(filepath) as csvfile:
        file = csv.reader(csvfile, delimiter=d)
        colIter, _ = itertools.tee(file)
        if not any(colIter):
            return None
        numCols = len(next(colIter))
        for i in range(numCols):
            results.append([])
        for row in file:
            for i in range(numCols):
                results[i].append(float(row[i]))
    return results


def plot_final_best(filePath):
    x, y, filepath, eqn = import_finalbest(filePath + "FinalBest.txt")
    plt.scatter(x, y, label=eqn)
    dataset = read_csv('inputs/data.txt', '\t')
    x = dataset[0]
    y = dataset[1]
    plt.scatter(x, y, label='dataset')
    plt.legend()
    plt.title(eqn)
    plt.show()


def plot_highest_fitness(filepath):
    file = read_csv(filepath + "HighestFitness.txt")
    if file is None:
        print("Can't plot highest fitness")
        return
    evals = file[0]
    fitnesses = file[1]
    plt.plot(evals, fitnesses)
    plt.title("Highest Fitness")
    plt.show()


def plot_diversity(dirpath):
    csvfile = read_csv(dirpath + "Diversity.txt")
    if csvfile is None:
        print("Can't plot diversity")
        return
    evaluations = csvfile[0]
    diversity = csvfile[1]
    plt.plot(evaluations, diversity)
    plt.ylabel('Diversity')
    plt.xlabel('Evaluations')
    plt.title("Diversity")
    plt.show()


def plot_fitness_dotplot(filepath):
    file = read_csv(filepath + "FitnessDotPlot.txt")
    if file is None:
        print("Can't plot dotplot")
        return
    evals = file[0]
    fitnesses = file[1]
    plt.scatter(evals, fitnesses, s=0.05)
    plt.title("Fitness Dot Plot")
    plt.show()


def plot_dir(dir):
    plot_final_best(dir)
    plot_fitness_dotplot(dir)
    plot_diversity(dir)
    plot_highest_fitness(dir)


def avg_learning_curve(runpaths, label):
    piecewises = []
    for runpath in runpaths:
        run = read_csv(runpath)
        piecewises.append((run[0], run[1]))
    minx = None
    for x, y in piecewises:
        if minx == None or minx > x[len(x)-1]:
            minx = int(x[len(x)-1])
    print("Sampling Piecewise from 0 to {}".format(minx))
    x = range(0, int(minx))
    y = []
    for _ in range(len(piecewises)):
        y.append([])

    p_i = 0
    for p_x, p_y in piecewises:
        for i in range(len(p_x) - 1):
            x0 = p_x[i]
            x1 = p_x[i+1]
            y0 = p_y[i]
            y1 = p_y[i+1]
            samples = np.linspace(y0, y1, num=(x1 - x0 + 1))
            y[p_i].extend(samples[:-1])
        p_i += 1

    for y_i in y:
        if len(y_i) < minx:
            minx = len(y_i)
    x = x[:minx]
    z = []
    for y_i in y:
        y_i = np.array(y_i[:minx])
        z.append(y_i[:minx])
    n_squared = math.sqrt(len(z))
    mean = np.mean(z, axis=0)
    std = np.std(z, axis=0) / n_squared
    l = min(len(mean), len(x))
    plt.plot(x[:l], mean[:l], label=label)
    plt.fill_between(x[:l], mean[:l]+std[:l], mean[:l]-std[:l], alpha=0.3)


rs_runs = ["runs/run45-rs/HighestFitness.txt",
           "runs/run46-rs/HighestFitness.txt",
           "runs/run47-rs/HighestFitness.txt",
           "runs/run48-rs/HighestFitness.txt",
           "runs/run49-rs/HighestFitness.txt"]

hc_runs = ["runs/run55-hc/HighestFitness.txt",
           "runs/run56-hc/HighestFitness.txt",
           "runs/run57-hc/HighestFitness.txt",
           "runs/run58-hc/HighestFitness.txt",
           "runs/run59-hc/HighestFitness.txt",
           "runs/run60-hc/HighestFitness.txt"]

ea_diversity_runs = ["runs/run61-ea-night/HighestFitness.txt",
                     "runs/run62-ea-night/HighestFitness.txt",
                     "runs/run63-ea-night/HighestFitness.txt",
                     "runs/run64-ea-night/HighestFitness.txt",
                     "runs/run65-ea-night/HighestFitness.txt",
                     "runs/run66-ea-night/HighestFitness.txt",
                     "runs/run67-ea-night/HighestFitness.txt",
                     "runs/run68-ea-night/HighestFitness.txt",  # potentially messed
up
                     "runs/run69-ea-night/HighestFitness.txt",
                     "runs/run70-ea-night/HighestFitness.txt"]

ea_tournament_runs = ["runs/run71-ea-tournament/HighestFitness.txt",
                      "runs/run72-ea-tournament/HighestFitness.txt",
                      "runs/run73-ea-tournament/HighestFitness.txt",
                      "runs/run74-ea-tournament/HighestFitness.txt",
                      "runs/run75-ea-tournament/HighestFitness.txt",
                      "runs/run77-ea-tournament/HighestFitness.txt",
                      "runs/run77-ea-tournament/HighestFitness.txt",
                      "runs/run78-ea-tournament/HighestFitness.txt",  # potentially
messed up
                      "runs/run79-ea-tournament/HighestFitness.txt",
                      "runs/run80-ea-tournament/HighestFitness.txt"]

gp_large_pop_runs = ["runs/run81-diverse-large-pop/HighestFitness.txt",
                     "runs/run82-diverse-large-pop/HighestFitness.txt",
                     "runs/run83-diverse-large-pop/HighestFitness.txt",
                     "runs/run84-diverse-large-pop/HighestFitness.txt"]
```

```
def plot_lc():
    avg_learning_curve(gp_large_pop_runs,
                       "GP Large Population")
    # avg_learning_curve(ea_diversity_runs,
    #                    "GP Diversity Selector")
    # avg_learning_curve(ea_tournament_runs,
    #                    "GP Tournament Selector")
    # avg_learning_curve(rs_runs, "Random Search")
    # avg_learning_curve(hc_runs, "Hill Climber")
    # plt.xscale("log")
    plt.xlabel("Evaluations")
    plt.ylabel("Fitness")
    plt.legend()
    plt.show()


tmp = [([0, 2, 6, 8, 10, 20], [0, 4, 4, 2, 0, 0, 0]),
       ([0, 4, 9, 10, 20], [0, 3, 3, 0, 0])]

if __name__ == "__main__":
    if(len(sys.argv) == 1):
        plot_lc()
        exit()
    # Parse Arguments
    parser = argparse.ArgumentParser(description='Plot a run')
    parser.add_argument(
        '-t', '--title', nargs='+', help='title of output graph', required=False)
    parser.add_argument('-r', '--runs', nargs='+', action='append',
                        help='runs to average', required=False)
    parser.add_argument('-l', '--labels', nargs='+',
                        help='labels for each group of run', required=False)
    args = parser.parse_args()
    dirname = "runs/tmp/"
    if args.runs != None:
        dirname = get_dir_name(args.runs[0][0])
    plot_dir(dirname)

# plot_learning_curve(args)
```

Reproducers

AsexualReproducer.hpp

```
#ifndef _ASEXUAL_REPRODUCER_HPP_
#define _ASEXUAL_REPRODUCER_HPP_
#include "Reproducer.hpp"
namespace SymbolicRegression
{
using namespace std;
class AsexualReproducer : public Reproducer
{
public:
    inline AsexualReproducer(int populationCount) : Reproducer(populationCount){};
    virtual shared_ptr<list<shared_ptr<Expression>>> Reproduce(
        const list<shared_ptr<Expression>> &parents) override;
    inline shared_ptr<Expression> CreateOffspring(const shared_ptr<Expression> p1,
const shared_ptr<Expression> p2) override
    {
        return Expression::GenerateRandomExpression(0, true);
    }
};
} // namespace SymbolicRegression

#endif
```

AsexualReproducer.cpp

```
#include "AsexualReproducer.hpp"
#include "../Solver.hpp"
#include "../mutators/ConstantMultiplierMutator.hpp"
#include "../mutators/ConstantMutator.hpp"
#include "../mutators/SubexpressionMutator.hpp"
#include "../mutators/TrigMultiplierMutator.hpp"
#include "../mutators/TruncateMutator.hpp"
namespace SymbolicRegression
{
shared_ptr<list<shared_ptr<Expression>>> AsexualReproducer::Reproduce(
    const list<shared_ptr<Expression>> &parents)
{
    m_offsprings.clear();
    int parentCount = int(parents.size());
    while (m_offsprings.size() < m_populationCount)
    {
        int idx = int(Expression::RandomF(0.0f, float(m_populationCount - 1)));
        auto parent = parents.begin();
        advance(parent, idx);
        shared_ptr<Expression> offspring;
        int attempts = 0;
        do
        {
            offspring = Expression::Copy(*parent);
            offspring = ConstantMutator::Mutate(offspring);
            if (Expression::RandomF() > 0.7f)
            {
                offspring = ConstantMultiplierMutator::Mutate(offspring);
            }
            if (Expression::RandomF() > 0.7f)
            {
                offspring = TrigMultiplierMutator::Mutate(offspring);
            }
            if (Expression::RandomF() > 0.7f)
            {
                offspring = SubexpressionMutator::Mutate(offspring);
            }
            if (Expression::RandomF() > 0.7f)
            {
                offspring = TruncateMutator::Mutate(offspring);
            }
            attempts += 1;
            if (attempts > 1000)
            {
                m_offsprings.insert(make_pair((*parent)->ToString(), *parent));
                break;
            }
        } while (offspring->Fitness() < (*parent)->Fitness() ||
                 Expression::RandomF() < Solver::GetTemp());
        if (Expression::IsValid(offspring) && m_offsprings.find(offspring-
>ToString()) == m_offsprings.end())
        {
            m_offsprings.insert(make_pair(offspring->ToString(), offspring));
        }
    }
    shared_ptr<list<shared_ptr<Expression>>> output(new list<shared_ptr<Expression>>
(m_offsprings.size()));
    transform(m_offsprings.begin(), m_offsprings.end(), output->begin(), [](auto p)
{
        return p.second;
    });
    return output;
}
} // namespace SymbolicRegression
```

CrossoverMutatorReproducer.hpp

```
#ifndef _CROSSOVER_MUTATOR_REPRODUCER_HPP_
#define _CROSSOVER_MUTATOR_REPRODUCER_HPP_
#include "Reproducer.hpp"
namespace SymbolicRegression
{
using namespace std;
class CrossoverMutatorReproducer : public Reproducer
{

public:
    inline CrossoverMutatorReproducer(int populationCount) :
Reproducer(populationCount){};
    virtual shared_ptr<Expression> CreateOffspring(const shared_ptr<Expression> p1,
const shared_ptr<Expression> p2) override;

private:
    shared_ptr<Expression> *FindCrossoverPoint(shared_ptr<Expression> root, int
maxDepth);
};
} // namespace SymbolicRegression

#endif
```

CrossoverMutatorReproducer.cpp

```
#include "CrossoverMutatorReproducer.hpp"
#include <memory>
#include "../../expression/Expression.hpp"
#include "../mutators/ConstantMutator.hpp"
#include "../mutators/SubexpressionMutator.hpp"
#include "../mutators/ConstantMultiplierMutator.hpp"
#include "../mutators/TrigMultiplierMutator.hpp"
#include "../mutators/TruncateMutator.hpp"
#include <algorithm>
#include <iostream>
#include "../../engine/Config.hpp"
#include "../Solver.hpp"
namespace SymbolicRegression
{
using namespace std;

shared_ptr<Expression>
*CrossoverMutatorReproducer::FindCrossoverPoint(shared_ptr<Expression> root, int
maxDepth)
{
    auto collapsedSubExp = root->Collapse(root);
    vector<shared_ptr<Expression>> operators;
    copy_if(
        collapsedSubExp->begin(),
        collapsedSubExp->end(),
        back_inserter(operators),
        [&](auto e) {
            return e->Order() > 0;
        });
    if (operators.size() == 0)
        return nullptr;
    auto expToChange = operators[int(Expression::RandomF(0, 100)) %
operators.size()];
    return &(expToChange->m_subexpressions[int(Expression::RandomF(0,
float(expToChange->Order()) - 1.0f))]);
}

shared_ptr<Expression> CrossoverMutatorReproducer::CreateOffspring(const
shared_ptr<Expression> p1, const shared_ptr<Expression> p2)
{
    shared_ptr<Expression> child;
    shared_ptr<Expression> bestChildSoFar;
    int attempts = 0;
    do
    {
        child = Expression::Copy(p1);
        // Not tall enough to crossover
        if (child->Depth() == 1 || p2->Depth() == 1)
        {
            return child;
        }

        // Find a random cross over point
        auto *cross1 = FindCrossoverPoint(child, Config::GetInt("MaxDepth"));
        auto *cross2 = FindCrossoverPoint(p2, (*cross1)->Depth());
        if (cross1 != nullptr && cross2 != nullptr)
        {
            *cross1 = Expression::Copy(*cross2);
        }
        child = ConstantMutator::Mutate(child);
        if (Expression::RandomF() > 0.8f)
        {
            child = ConstantMultiplierMutator::Mutate(child);
        }
        if (Expression::RandomF() > 0.8f)
        {
            child = TrigMultiplierMutator::Mutate(child);
        }
        if (Expression::RandomF() > 0.8f)
        {
            child = SubexpressionMutator::Mutate(child);
        }
        if (Expression::RandomF() > 0.8f)
        {
            child = TruncateMutator::Mutate(child);
        }
        child = Expression::Simplify(child);

        if (Expression::RandomF() < Solver::GetTemp())
        {
            return child;
        }
        if (!bestChildSoFar || child->Fitness() > bestChildSoFar->Fitness())
        {
            bestChildSoFar = child;
        }
        attempts += 1;
        if (attempts > 100)
        {
            return bestChildSoFar;
        }
    } while (bestChildSoFar->Fitness() < p1->Fitness() &&
             bestChildSoFar->Fitness() < p2->Fitness());
    return bestChildSoFar;
}

} // namespace SymbolicRegression
```

MutatorReproducer.hpp

```
#ifndef _MUTATOR_REPRODUCER_HPP_
#define _MUTATOR_REPRODUCER_HPP_
#include "Reproducer.hpp"
namespace SymbolicRegression
{
using namespace std;
class MutatorReproducer : public Reproducer
{
public:
    inline MutatorReproducer(int populationCount) : Reproducer(populationCount){};
    virtual shared_ptr<Expression> CreateOffspring(const shared_ptr<Expression> p1,
const shared_ptr<Expression> p2) override;
};
} // namespace SymbolicRegression

#endif
```

MutatorReproducer.cpp

```
#include "RandomReproducer.hpp"
#include <algorithm>
#include "MutatorReproducer.hpp"
#include <iostream>
#include "../mutators/ConstantMutator.hpp"
#include "../mutators/SubexpressionMutator.hpp"
#include "../mutators/TruncateMutator.hpp"
#include "../Solver.hpp"
namespace SymbolicRegression
{
using namespace std;

shared_ptr<Expression> MutatorReproducer::CreateOffspring(const
shared_ptr<Expression> p1, const shared_ptr<Expression> p2)
{
    shared_ptr<Expression> child;
    // Copy parent 1
    float parentFitness = p2->Fitness();
    // Mutate child
    do
    {
        if (Expression::RandomF() > 0.5f)
        {
            child = Expression::Copy(p2);
        }
        else
        {
            child = Expression::Copy(p1);
        }

        child = SubexpressionMutator::Mutate(child);
        if (Expression::RandomF() > 0.5f)
        {
            child = TruncateMutator::Mutate(child);
        }
        else
        {
            child = ConstantMutator::Mutate(child);
        }
        child = Expression::Simplify(child);
        if (Expression::RandomF() < Solver::GetTemp())
        {
            return child;
        }
    } while (child->Fitness() < parentFitness);

    return child;
}
} // namespace SymbolicRegression
```

RandomReproducer.hpp

```
#ifndef _RANDOM_REPRODUCER_HPP_
#define _RANDOM_REPRODUCER_HPP_
#include "Reproducer.hpp"
namespace SymbolicRegression
{
using namespace std;
class RandomReproducer : public Reproducer
{
public:
    inline RandomReproducer(int populationCount) : Reproducer(populationCount){};
    virtual shared_ptr<Expression> CreateOffspring(const shared_ptr<Expression> p1,
const shared_ptr<Expression> p2) override;
};
} // namespace SymbolicRegression

#endif
```

RandomReproducer.cpp

```
#include "RandomReproducer.hpp"
#include <algorithm>

namespace SymbolicRegression
{
using namespace std;

shared_ptr<Expression> RandomReproducer::CreateOffspring(const
shared_ptr<Expression> p1, const shared_ptr<Expression> p2)
{
    return Expression::GenerateRandomExpression(0, true);
}
} // namespace SymbolicRegression
```

Reproducer.hpp

```
#ifndef _REPRODUCER_HPP_
#define _REPRODUCER_HPP_
#include <memory>
```

```
#include <list>
#include <map>
#include <string>
#include <tuple>
#include "../../expression/Expression.hpp"
namespace SymbolicRegression
{
using namespace std;
class Reproducer
{
public:
    Reproducer(int populationCount);
    shared_ptr<list<shared_ptr<Expression>>> AsyncReproduce(
        const list<shared_ptr<Expression>> &parents);
    virtual shared_ptr<list<shared_ptr<Expression>>> Reproduce(
        const list<shared_ptr<Expression>> &parents);
    virtual shared_ptr<Expression> CreateOffspring(
        const shared_ptr<Expression> p1, const shared_ptr<Expression> p2) = 0;

protected:
    int m_populationCount = -1;
    map<string, shared_ptr<Expression>> m_offsprings;
    void TryInsertOffspring(shared_ptr<Expression> exp);

private:
    tuple<shared_ptr<Expression>, shared_ptr<Expression>> ChooseTwoParents(const
list<shared_ptr<Expression>> &parents);

private:
    bool m_stop;
    int m_minThreads = -1;
    int m_maxThreads = -1;
};
} // namespace SymbolicRegression

#endif
```

Reproducer.cpp

```
#include "Reproducer.hpp"
#include <mutex>
#include <future>
#include <deque>
#include <algorithm>
#include "../engine/Config.hpp"
namespace SymbolicRegression
{
using namespace std;
```

```
mutex mu;

Reproducer::Reproducer(int populationCount)
{
    m_populationCount = populationCount;
    m_minThreads = Config::GetInt("MinThreads");
    m_maxThreads = Config::GetInt("MaxThreads");
}

void Reproducer::TryInsertOffspring(shared_ptr<Expression> exp)
{
    if (exp->Depth() > Config::GetInt("MaxDepth"))
    {
        cout << "Invalid Expression of depth " << exp->Depth() << endl;
        return;
    }
    lock_guard<mutex> lock(mu);
    if (m_stop)
        return;
    // insert offspring if it is unique
    if (Expression::IsValid(exp) && m_offsprings.find(exp->ToString()) ==
m_offsprings.end())
    {
        m_offsprings.insert(make_pair(exp->ToString(), exp));
    }
    if (m_offsprings.size() >= m_populationCount)
    {
        m_stop = true;
    }
}

tuple<shared_ptr<Expression>, shared_ptr<Expression>>
Reproducer::ChooseTwoParents(const list<shared_ptr<Expression>> &parents)
{
    int parentCount = int(parents.size());
    if (parentCount < 2)
        return make_tuple(parents.front(), parents.front());
    int idx1 = int(Expression::RandomF(0, 1000000)) % parentCount;
    int idx2 = int(Expression::RandomF(0, 1000000)) % parentCount;
    while (true)
    {
        idx2 = (idx2 + 1) % parentCount;
        auto p1 = parents.begin();
        advance(p1, idx1);
        auto p2 = parents.begin();
        advance(p2, idx2);
        if ((*p1)->ToString() != (*p2)->ToString())
        {
            return make_tuple(*p1, *p2);
        }
    }
}

shared_ptr<list<shared_ptr<Expression>>> Reproducer::AsyncReproduce(const
list<shared_ptr<Expression>> &parents)
{
    m_offsprings.clear();
    m_stop = false;
    while (!m_stop)
    {
        size_t n = max(m_populationCount - (int)m_offsprings.size(), m_minThreads);
        if (m_maxThreads > 0)
            n = min(int(n), m_maxThreads);
        vector<future<void>> tasks;
        tasks.reserve(n);
        while (tasks.size() < tasks.capacity())
        {
            auto pair = ChooseTwoParents(parents);
            // queue up asynchronous reproduce task
            tasks.push_back(async([&]() {
                TryInsertOffspring(CreateOffspring(get<0>(pair), get<1>(pair)));
            }));
        }
        for_each(tasks.begin(), tasks.end(), [](future<void> &task) {
            task.get();
        });
    }
    shared_ptr<list<shared_ptr<Expression>>> output(new list<shared_ptr<Expression>>
(m_offsprings.size()));
    transform(m_offsprings.begin(), m_offsprings.end(), output->begin(), [](auto p)
{
        return p.second;
    });

    return output;
}

shared_ptr<list<shared_ptr<Expression>>> Reproducer::Reproduce(const
list<shared_ptr<Expression>> &parents)
{
    m_offsprings.clear();
    int parentCount = int(parents.size());
    while (m_offsprings.size() < m_populationCount)
    {
        auto pair = ChooseTwoParents(parents);
        auto offspring = CreateOffspring(get<0>(pair), get<1>(pair));
        if (Expression::IsValid(offspring) && m_offsprings.find(offspring-
>ToString()) == m_offsprings.end())
        {
            m_offsprings.insert(make_pair(offspring->ToString(), offspring));
        }
    }

    shared_ptr<list<shared_ptr<Expression>>> output(new list<shared_ptr<Expression>>
(m_offsprings.size()));
    transform(m_offsprings.begin(), m_offsprings.end(), output->begin(), [](auto p)
{
        return p.second;
    });
    return output;
}
} // namespace SymbolicRegression
```

Selectors

DiversitySelector.hpp

```
#ifndef _DIVERSITY_SELECTOR_HPP_
#define _DIVERSITY_SELECTOR_HPP_
#include "Selector.hpp"
#include "TournamentSelector.hpp"
namespace SymbolicRegression
{
class DiversitySelector : public Selector
{
public:
    inline DiversitySelector() : Selector()
    {
        m_tournamentSelector = TournamentSelector();
    }
    virtual tuple<shared_ptr<Expression>, shared_ptr<Expression>> Select(
        const list<shared_ptr<Expression>> &population) override;

private:
    TournamentSelector m_tournamentSelector;
};
} // namespace SymbolicRegression

#endif
```

DiversitySelector.cpp

```
#include "DiversitySelector.hpp"
#include "../../expression/Expression.hpp"
namespace SymbolicRegression
{
tuple<shared_ptr<Expression>, shared_ptr<Expression>> DiversitySelector::Select(
    const list<shared_ptr<Expression>> &population)
{
    float threshold = 15.0f;
    float decay = 0.95f;
    int attempt = 0;
    while (true)
    {
        auto [p1, p2] = m_tournamentSelector.Select(population);
        if (Expression::Diversity(p1, p2) > threshold || attempt > 50)
        {
            return make_tuple(p1, p2);
        }
        attempt += 1;
        threshold *= decay;
    }
}
} // namespace SymbolicRegression
```

NichingSelector.hpp

```
#ifndef _NICHING_SELECTOR_HPP_
#define _NICHING_SELECTOR_HPP_
#include "Selector.hpp"
#include "TournamentSelector.hpp"
namespace SymbolicRegression
{
class NichingSelector : public Selector
{
public:
    inline NichingSelector() : Selector()
    {
        m_tournamentSelector = TournamentSelector();
    }
    virtual tuple<shared_ptr<Expression>, shared_ptr<Expression>> Select(
        const list<shared_ptr<Expression>> &population) override;

protected:
    TournamentSelector m_tournamentSelector;
};
} // namespace SymbolicRegression

#endif
```

NichingSelector.cpp

```
#include "NichingSelector.hpp"
#include "../../expression/Expression.hpp"
namespace SymbolicRegression
{
tuple<shared_ptr<Expression>, shared_ptr<Expression>> NichingSelector::Select(
    const list<shared_ptr<Expression>> &population)
{
    float threshold = 2.0f;
    float growth = 1.05f;
    int attempt = 0;
    while (true)
    {
        cout << "\tNiching trying " << threshold << endl;
        auto [p1, p2] = m_tournamentSelector.Select(population);
        if (Expression::Diversity(p1, p2) < threshold || attempt > 50)
        {
            cout << "Niching DONE " << threshold << endl;
            return make_tuple(p1, p2);
        }
        attempt += 1;
        threshold *= growth;
    }
}
} // namespace SymbolicRegression
```

Selector.hpp

```
#ifndef _SELECTOR_HPP_
#define _SELECTOR_HPP_
#include "../../expression/Expression.hpp"
#include <memory>
#include <tuple>
namespace SymbolicRegression
{
using namespace std;
class Selector
{
public:
    virtual tuple<shared_ptr<Expression>, shared_ptr<Expression>> Select(
        const list<shared_ptr<Expression>> &population) = 0;
};
} // namespace SymbolicRegression

#endif
```

TournamentSelector.hpp

```
#ifndef _TOURNAMENT_SELECTOR_HPP_
#define _TOURNAMENT_SELECTOR_HPP_
#include "Selector.hpp"
namespace SymbolicRegression
{
class TournamentSelector : public Selector
{
public:
    TournamentSelector();
    virtual tuple<shared_ptr<Expression>, shared_ptr<Expression>> Select(
        const list<shared_ptr<Expression>> &population) override;

protected:
    int m_numPlayers = -1;
    shared_ptr<Expression> DoTournament(const list<shared_ptr<Expression>>
&population);
};
} // namespace SymbolicRegression

#endif
```

TournamentSelector.cpp

```

#include "TournamentSelector.hpp"
#include "../../expression/Expression.hpp"
#include "../../engine/Config.hpp"
namespace SymbolicRegression
{

TournamentSelector::TournamentSelector() : Selector()
{
    m_numPlayers = Config::GetInt("TournamentPlayersCount");
}

shared_ptr<Expression> TournamentSelector::DoTournament(const
list<shared_ptr<Expression>> &population)
{
    int populationCount = int(population.size());
    vector<shared_ptr<Expression>> players;
    if (m_numPlayers == population.size())
    {
        return *max_element(population.begin(), population.end(),
Expression::FitnessComparer);
    }
    else if (m_numPlayers > population.size())
    {
        cout << "ERROR:PARENTS LESS THAN TOURNAMNET SIZE" << endl;
        throw exception();
    }
    while (players.size() < m_numPlayers)
    {
        auto it = population.begin();
        int idx = int(Expression::RandomF(0.0f, float(populationCount - 1)));
        advance(it, idx);

        bool notunique = any_of(players.begin(), players.end(), [&](auto p) {
            return p->ToString() == (*it)->ToString();
        });
        if (!notunique)
        {
            players.push_back(*it);
        }
    }
    sort(players.begin(), players.end(), Expression::FitnessComparer);
    return players[0];
}

tuple<shared_ptr<Expression>, shared_ptr<Expression>> TournamentSelector::Select(
    const list<shared_ptr<Expression>> &population)
{
    if (population.size() == 2)
    {
        return make_tuple(population.front(), population.back());
    }
    else if (population.size() < 2)
    {
        cout << "ERROR: POPULATION TOO SMALL" << endl;
        throw exception();
    }
    list<shared_ptr<Expression>> parents = list<shared_ptr<Expression>>
(population.begin(), population.end());
    shared_ptr<Expression> p2,
        p1 = DoTournament(parents);
    parents.remove(p1);
    do
    {
        p2 = DoTournament(parents);
        parents.remove(p2);
    } while (p2->ToString() == p1->ToString());
    return make_tuple(p1, p2);
}
} // namespace SymbolicRegression
```

Solver

Solver.hpp

```
#ifndef _SOLVER_HPP_
#define _SOLVER_HPP_
#include <list>
#include <memory>
#include <mutex>
#include "reproducers/Reproducer.hpp"
#include "selectors/Selector.hpp"
#include "../engine/OutputLogger.hpp"
#include <ctime>
#include <chrono>
namespace SymbolicRegression
{
using namespace std;
class Solver
{
public:
    Solver();
    virtual void Run() = 0;
    void SaveOutput();
    static shared_ptr<Solver> Instance();
    static float GetTemp();
    static void DecayTemp();
    void SavePopulationFitnesses();
    float PopulationDiversity();
    inline bool ShouldStop()
    {
        return m_prevBestEvaluations < OutputLogger::GetEvaluations() / 2;
    }

protected:
    void InitializePopulation();
    void PrintPopulation();

protected:
    list<shared_ptr<Expression>> m_population;
    int m_populationCount = -1;
    shared_ptr<Reproducer> m_reproducer;
    int m_eliteCount = -1;
    float m_prevHighestFitness = -1;
    int m_prevBestEvaluations = -1;
    shared_ptr<Expression> m_prevBest;
    shared_ptr<Selector> m_selector;

private:
    static shared_ptr<Solver> m_instance;
    static float m_temperature;
    static mutex tempMutex;
    bool collectDataForDotPlot;
};
} // namespace SymbolicRegression
#endif
```

Solver.cpp

```

#include "Solver.hpp"
#include "../expression/Expression.hpp"
#include <functional>
#include <algorithm>
#include "../engine/Config.hpp"
#include "../engine/OutputLogger.hpp"
#include "reproducers/RandomReproducer.hpp"
#include "reproducers/MutatorReproducer.hpp"
#include "reproducers/AsexualReproducer.hpp"
#include "reproducers/CrossoverMutatorReproducer.hpp"
#include "selectors/TournamentSelector.hpp"
#include "selectors/DiversitySelector.hpp"
#include "selectors/NichingSelector.hpp"
#include <fstream>
#include <windows.h>
#include "../expression/Constant.hpp"
#include <cmath>
#include "GenerationalSolver.hpp"
#include "ContinuousSolver.hpp"
namespace SymbolicRegression
{
using namespace std;
shared_ptr<Solver> Solver::m_instance;
float Solver::m_temperature;
mutex Solver::tempMutex;
Solver::Solver()
{
    m_populationCount = Config::GetInt("PopulationCount");
    m_eliteCount = Config::GetInt("ElitesCount");
    m_temperature = Config::GetFloat("Init_T");
    string reproducerConfig = Config::GetString("Reproducer");
    if (reproducerConfig == "CrossoverMutator")
    {
        m_reproducer = shared_ptr<Reproducer>(new
CrossoverMutatorReproducer(m_populationCount));
    }
    else if (reproducerConfig == "Mutator")
    {
        m_reproducer = shared_ptr<Reproducer>(new
MutatorReproducer(m_populationCount));
    }
    else if (reproducerConfig == "Asexual")
    {
        m_reproducer = shared_ptr<Reproducer>(new
AsexualReproducer(m_populationCount));
    }
    else
    {
        m_reproducer = shared_ptr<Reproducer>(new
RandomReproducer(m_populationCount));
    }
    string selectorConfig = Config::GetString("Selector");
    if (selectorConfig == "Diversity")
    {
        m_selector = shared_ptr<Selector>(new DiversitySelector());
    }
    else if (selectorConfig == "Niching")
    {
        m_selector = shared_ptr<Selector>(new NichingSelector());
    }
    else
    {
        m_selector = shared_ptr<Selector>(new TournamentSelector());
    }

    collectDataForDotPlot = Config::GetInt("DotPlot");
}

shared_ptr<Solver> Solver::Instance()
{
    if (!m_instance)
    {
        string solverConfig = Config::GetString("Solver");
        if (solverConfig == "Continuous")
        {
            m_instance = shared_ptr<Solver>(new ContinuousSolver());
        }
        else
        {
            m_instance = shared_ptr<Solver>(new GenerationalSolver());
        }
    }
    return m_instance;
}

void Solver::PrintPopulation()
{
    for_each(m_population.begin(), m_population.end(), [](auto exp) {
        cout << exp->ToString() << " | ";
    });
    cout << endl;
}
void Solver::InitializePopulation()
{
    while (m_population.size() < m_populationCount)
    {
        auto newExp = Expression::GenerateRandomExpression(0, true);
        if (Expression::IsValid(newExp) && !any_of(m_population.begin(),
m_population.end(), [&](const shared_ptr<Expression> &exp) {
                return exp->ToString() == newExp->ToString();
            }))
        {
            m_population.emplace_front(newExp);
        }
    }
}

float Solver::PopulationDiversity()
{
    int n = int(m_population.size());
    float diversitySum = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            auto e1 = m_population.begin();
            advance(e1, i);
            auto e2 = m_population.begin();
            advance(e2, j);
            diversitySum += Expression::Diversity(*e1, *e2);
        }
    }
    return diversitySum * 0.0001f / n * n;
}

void Solver::SaveOutput()
{
    OutputLogger::Clear("FinalBest");
    OutputLogger::Log("FinalBest", m_prevBest->ToString());
    auto f = m_prevBest->ToFunction();
    for (float x = 0; x < 10; x += 0.001f)
    {
        OutputLogger::Log("FinalBest", to_string(x) + " " + to_string(f(x)));
    }
    string dirpath = Config::GetString("OutputPath");
    LPCSTR w_dirpath = LPCSTR(dirpath.c_str());
    //  wstring(dirpath.begin(), dirpath.end()).c_str();
    if (CreateDirectory(w_dirpath, NULL) || ERROR_ALREADY_EXISTS == GetLastError())
    {
        // Directory created
        auto keys = OutputLogger::GetKeys();
        for (auto &key : *keys)
        {
            ofstream f;
            f.open(dirpath + key + ".txt");
            if (f.is_open())
            {
                f << OutputLogger::Get(key);
            }
            else
            {
                cout << "Error opening " << dirpath + key + ".txt" << endl;
            }
            f.close();
        }
    }
    else
    {
        // Failed for some other reason
        cout << "Failed to create output directory" << endl;
        exit(-1);
    }
}

void Solver::SavePopulationFitnesses()
{
    string evals = to_string(OutputLogger::GetEvaluations());
    for_each(m_population.begin(), m_population.end(), [&evals](auto e) {
        OutputLogger::Log("FitnessDotPlot", evals + "," + to_string(e->Fitness()));
    });
}

float Solver::GetTemp()
{
    lock_guard<mutex> lock(tempMutex);
    return m_temperature;
}

void Solver::DecayTemp()
{
    lock_guard<mutex> lock(tempMutex);
    m_temperature *= Config::GetFloat("T_decay");
}
} // namespace SymbolicRegression
```

GenerationalSolver.hpp

```
#ifndef _GENERATIONAL_SOLVER_HPP_
#define _GENERATIONAL_SOLVER_HPP_
#include "Solver.hpp"
namespace SymbolicRegression
{
class GenerationalSolver : public Solver
{
public:
    virtual void Run() override;

protected:
    void Evolve();
};
} // namespace SymbolicRegression

#endif
```

GenerationalSolver.cpp

```
#include "GenerationalSolver.hpp"
#include "../engine/Config.hpp"
#include <algorithm>
#include <iterator>
#include "../engine/OutputLogger.hpp"
namespace SymbolicRegression
{

void GenerationalSolver::Run()
{
    int generationCount = Config::GetInt("GenerationCount");
    int saveEval = 0;
    InitializePopulation();
    for (int i = 0; i < generationCount; i++)
    {
        Evolve();
        DecayTemp();
        if (OutputLogger::GetEvaluations() > saveEval * 100000)
        {
            SaveOutput();
            saveEval++;
        }
    }
    SaveOutput();
    m_population.sort(Expression::FitnessComparer);
    auto finalBest = *m_population.begin();
    cout << "FITNESS " << m_prevHighestFitness
         << " after " << OutputLogger::GetEvaluations() << " evalutions at temp " <<
```

```
GetTemp() << endl;
    cout << "\t" + finalBest->ToString() << endl;
}

void GenerationalSolver::Evolve()
{
    for_each(m_population.begin(), m_population.end(),
             [](const shared_ptr<Expression> &exp) { exp->Fitness(); });
    // Sort in decreasing order of fitness
    m_population.sort(Expression::FitnessComparer);
    // Find best in current population
    shared_ptr<Expression> bestExpression = *(m_population.begin());
    if (bestExpression->Fitness() > m_prevHighestFitness)
    {
        m_prevHighestFitness = bestExpression->Fitness();
        cout << "FITNESS " << m_prevHighestFitness
             << " after " << OutputLogger::GetEvaluations() << " evalutions at temp
" << GetTemp() << endl;
        cout << "\t" + bestExpression->ToString() << endl;
    }
    // Selection
    auto it = m_population.begin();
    advance(it, m_population.size() * 0.7f);
    m_population.erase(it, m_population.end());
    // Reproduce
    // auto offspring = m_reproducer->AsyncReproduce(m_population);
    auto offspring = m_reproducer->Reproduce(m_population);

    // Handle Elites
    auto eliteEnd = m_population.begin();
    advance(eliteEnd, m_eliteCount);
    list<shared_ptr<Expression>> elites(m_population.begin(), eliteEnd);
    // Create new population
    m_population.clear();
    copy(elites.begin(), elites.end(), front_inserter(m_population));
    copy(offspring->begin(), offspring->end(), front_inserter(m_population));
    m_population.sort(Expression::FitnessComparer);

    // Ensure size of population
    if (m_population.size() < m_populationCount)
    {
        cout << "ERROR: LOST EXPRESSIONS" << endl;
        throw exception("Lost Expressions");
    }
    else if (m_population.size() > m_populationCount)
    {
        m_population.resize(m_populationCount);
    }
    // Collect Stats
    OutputLogger::Log("HighestFitness", to_string(OutputLogger::GetEvaluations()) +
" " + to_string(m_prevHighestFitness));
}
}
```

ContinuousSolver.hpp

```
#ifndef _CONTINUOUS_SOLVER_HPP_
#define _CONTINUOUS_SOLVER_HPP_
#include "Solver.hpp"
namespace SymbolicRegression
{
class ContinuousSolver : public Solver
{
public:
    virtual void Run() override;

protected:
    void EvolveRound();
};
} // namespace SymbolicRegression

#endif
```

ContinuousSolver.cpp

```
#include "ContinuousSolver.hpp"
#include "../engine/Config.hpp"
#include "../engine/OutputLogger.hpp"
#include <chrono>
#include <thread>
namespace SymbolicRegression
{
using namespace std;

void ContinuousSolver::EvolveRound()
{
    if (typeid(*m_reproducer).name() == string("class
SymbolicRegression::RandomReproducer") ||
        typeid(*m_reproducer).name() == string("class
SymbolicRegression::AsexualReproducer"))
    {
        auto offsprings = m_reproducer->Reproduce(m_population);
        m_population.clear();
        for_each(offsprings->begin(), offsprings->end(), [&](auto e) {
            m_population.emplace_front(e);
        });
    }
    else
    {
        // select two parents using selector
        auto [p1, p2] = m_selector->Select(m_population);
        // remove parents from population
        // reproduce the two parents, giving at least 2 individuals back as a list
        auto offspring = m_reproducer->CreateOffspring(p1, p2);
        bool notunique = any_of(m_population.begin(), m_population.end(), [&](auto
p) {
            return p->ToString() == offspring->ToString();
        });
        if (!Expression::IsValid(offspring) || notunique)
        {
            return;
        }
        m_population.emplace_front(offspring);
        // remove all invalid expressions
        // handle over population
        m_population.sort(Expression::FitnessComparer);
        while (m_population.size() > m_populationCount)
        {
            m_population.pop_back();
        }
    }
    OutputLogger::Log("HighestFitness",
                      to_string(OutputLogger::GetEvaluations()) +
                          "," + to_string(m_prevHighestFitness));
}

void ContinuousSolver::Run()
{
    m_prevHighestFitness = -1;
    int saveEval = 0;
    InitializePopulation();
    int round = 0;
    // Run for at least 6000 rounds
    // Afterwards, if double evaluations without getting better, quit
    while (round < 6000 || !ShouldStop())
    {
        EvolveRound();
        for_each(m_population.begin(), m_population.end(), [](auto e) {
            e = Expression::Simplify(e);
        });
        m_population.sort(Expression::FitnessComparer);
        if ((*m_population.begin())->Fitness() > m_prevHighestFitness)
        {
            m_prevBest = (*m_population.begin());
            m_prevHighestFitness = m_prevBest->Fitness();
            m_prevBestEvaluations = OutputLogger::GetEvaluations();
            cout << "FITNESS: " << m_prevHighestFitness << " | Temp " << GetTemp()
<< endl;
            cout << "\t" << m_prevBest->ToString() << endl;
            SaveOutput();
        }
        else if (OutputLogger::GetEvaluations() > saveEval * 100)
        {
            SaveOutput();
            saveEval++;
        }
        DecayTemp();
        SavePopulationFitnesses();
        OutputLogger::Log("Diversity",
                          to_string(OutputLogger::GetEvaluations()) +
                              "," + to_string(PopulationDiversity()));
        round += 1;
    }
    SaveOutput();
    cout << "FITNESS " << m_prevHighestFitness
         << " after " << OutputLogger::GetEvaluations() << " evalutions at temp " <<
GetTemp() << endl;
    cout << "\t" + m_prevBest->ToString() << endl;
}
} // namespace SymbolicRegression
```

Expression

Expression.hpp

```
#ifndef _EXPRESSION_HPP_
#define _EXPRESSION_HPP_
#include <functional>
#include <vector>
#include <string>
#include <iostream>
#include <memory>
#include <algorithm>
#include <mutex>
#include <typeinfo>
namespace SymbolicRegression
{
using namespace std;
class Expression
{
public:
    // Factory Functions
    shared_ptr<Expression> static GenerateRandomExpression(int level, bool
noConstant = false, bool noZero = false, bool noTrig = false);
    shared_ptr<Expression> static GenerateRandomZeroOrderExpression(int level);
    shared_ptr<Expression> static GenerateRandomTrigExpression(int level);
    shared_ptr<Expression> static GenerateRandomBinaryOperator(int level);

    friend class SubexpressionMutator;
    friend class TruncateMutator;
    friend class ConstantMultiplierMutator;
    friend class TrigMultiplierMutator;
    friend class CrossoverMutatorReproducer;

public:
    /*
        STATIC HELPER FUNCTIONS
    */
    static function<bool(const shared_ptr<Expression> &, const
shared_ptr<Expression> &)> FitnessComparer;

    static shared_ptr<Expression> Initialize(shared_ptr<Expression> self,
shared_ptr<Expression> parent);

    shared_ptr<vector<shared_ptr<Expression>>> Collapse(const shared_ptr<Expression>
&self) const;
    static shared_ptr<Expression> Simplify(shared_ptr<Expression> exp);

    static float RandomF();
    static float RandomF(float min, float max);
    inline virtual string ToString() const = 0;
    static shared_ptr<Expression> Copy(const shared_ptr<Expression> &source);

public:
    /*
        PUBLIC ACCESSORS
    */
    virtual function<float(float)> ToFunction() const;

    float Fitness();
    float Fitness(function<float(float)> f);
    float CalculateFitness(function<float(float)> f) const;

    inline int Order() const { return m_order; }
    inline int Level() const { return m_level; }
    int Depth() const;
    static bool IsValid(shared_ptr<Expression> exp);

public:
    /*
    Diversity
    */
    static float Diversity(const shared_ptr<Expression> &e1, const
shared_ptr<Expression> &e2);

public:
    /*
        EXPRESSION PREDICATES TO HELP WITH EXPRESSION SIMPLIFICATION
    */
    typedef function<bool(const shared_ptr<Expression> &)> ExpressionPredicate;
    static ExpressionPredicate evaluatesToConstant;
    static ExpressionPredicate subexpressionsCancelOut;
    static ExpressionPredicate all;
    static ExpressionPredicate isTrigFunction;
    static ExpressionPredicate minusOrDivide;
    static function<float(float)> one;
#define EXPRESSION_TYPE(exp) string(typeid(*exp).name())
#define CONSTANT_T string("class SymbolicRegression::Constant")
#define VARIABLE_T string("class SymbolicRegression::Variable")
#define SIN_T string("class SymbolicRegression::Sin")
#define COS_T string("class SymbolicRegression::Cos")
#define PLUS_T string("class SymbolicRegression::Plus")
#define MINUS_T string("class SymbolicRegression::Minus")
#define DIVIDE_T string("class SymbolicRegression::Divide")
#define MULTIPLY_T string("class SymbolicRegression::Multiply")

protected:
    Expression(int level);
    Expression(const Expression &other);
    inline static vector<shared_ptr<Expression>>::iterator FindFirst(
        vector<shared_ptr<Expression>> &operators,
        ExpressionPredicate subexpPredicate,
        ExpressionPredicate opPredicate)
    {
        return find_if(operators.begin(), operators.end(), [&](auto op) {
            return opPredicate(op) && all_of(op->m_subexpressions.begin(), op-
>m_subexpressions.end(), subexpPredicate);
        });
    }

    inline static bool AnySubExpression(shared_ptr<Expression> exp,
ExpressionPredicate pred)
    {
        return pred(exp) || any_of(
                                exp->m_subexpressions.begin(),
                                exp->m_subexpressions.end(), [&](auto subexp) {
                                    return AnySubExpression(subexp, pred);
                                });
    }

    // replaces e1 with e2 in e1's original expression tree

    inline static void ReplaceExpression(shared_ptr<Expression> e1,
shared_ptr<Expression> e2)
    {
        if (!e1->m_parent.expired())
        {
            auto parent = e1->m_parent.lock();
            for (int i = 0; i < parent->m_subexpressions.size(); i++)
            {
                if (parent->m_subexpressions[i]->ToString() == e1->ToString())
                {
                    parent->m_subexpressions[i] = e2;
                }
            }
        }
    }

    inline void RecalculateLevels(int level)
    {
        m_level = level;
        for_each(m_subexpressions.begin(), m_subexpressions.end(), [&](auto subexp)
{
            subexp->RecalculateLevels(level + 1);
        });
    }

protected:
    int m_order = -1;                    // How many parameters the current expression
needs
    function<float(float)> m_func = 0; //function presenting this expression node's
function
    weak_ptr<Expression> m_parent;
    vector<shared_ptr<Expression>> m_subexpressions;
    float m_fitness = -1;
    int m_level = -1;

private:
    static mutex randMutex;
};
} // namespace SymbolicRegression
// ostream &operator<<(ostream &os, const Expression &e)
// {
//     os << e.ToString();
//     return os;
// }
#endif
```

Expression.cpp

```
#include "Expression.hpp"
#include <iostream>
#include "Constant.hpp"
#include "Cos.hpp"
#include "Sin.hpp"
#include "Plus.hpp"
#include "Minus.hpp"
#include "Multiply.hpp"
#include "Divide.hpp"
#include "Variable.hpp"
#include "../engine/Config.hpp"
#include <functional>
#include "../engine/OutputLogger.hpp"
#include <typeinfo>
#include "../genetic-programming/Solver.hpp"
#include <chrono>
#include <thread>
namespace SymbolicRegression
{
using namespace std;

#pragma region Static Variables
function<bool(
    const shared_ptr<Expression> &,
    const shared_ptr<Expression> &)>
    Expression::FitnessComparer = [](const shared_ptr<Expression> &a,
                                     const shared_ptr<Expression> &b) -> bool {
    return a->Fitness() > b->Fitness();
};

Expression::ExpressionPredicate Expression::evaluatesToConstant = [](auto
subexpression) {
    auto f = subexpression->ToFunction();
    float f0 = f(0);
    float f1 = f(1);
    float f2 = f(2);
    if (abs(f0 - f1) < 0.1f && abs(f2 - f1) < 0.1f)
        return true;
    else
        return false;
};

Expression::ExpressionPredicate Expression::subexpressionsCancelOut = [](auto
subexpression) {
    return EXPRESSION_TYPE(subexpression) == VARIABLE_T;
};

Expression::ExpressionPredicate Expression::all = [](auto op) {
    return true;
};

Expression::ExpressionPredicate Expression::minusOrDivide = [&](auto op) {
    auto typeStr = EXPRESSION_TYPE(op);
    return typeStr == DIVIDE_T ||
           typeStr == MINUS_T;
};

Expression::ExpressionPredicate Expression::isTrigFunction = [&](auto op) {
    auto typeStr = EXPRESSION_TYPE(op);
    return typeStr == SIN_T ||
           typeStr == SIN_T;
};

function<float(float)> Expression::one = [](float x) { return 1.0f; };
mutex Expression::randMutex;
#pragma endregion

Expression::Expression(int level)
{
    m_func = 0;
    m_order = -1;
    m_level = level;
}

Expression::Expression(const Expression &other)
```

```
{
    m_order = other.m_order;
    m_func = 0;
    m_level = other.m_level;
    for (int i = 0; i < other.m_subexpressions.size(); i++)
    {
        m_subexpressions.push_back(Copy(other.m_subexpressions[i]));
    }
}

float Expression::Fitness()
{
    if (m_fitness == -1)
    {
        m_fitness = Fitness(one);
    }
    return m_fitness;
}

float Expression::Fitness(function<float(float)> f)
{
    return CalculateFitness(f);
}

float Expression::CalculateFitness(function<float(float)> f2) const
{
    using namespace std;
    function<float(float)> f = ToFunction();
    float AbsoluteErrorSum = 0;
    for_each(Config::Data->begin(), Config::Data->end(), [&](tuple<float, float> x)
{
        AbsoluteErrorSum += abs(f(get<0>(x)) * f2(get<0>(x)) - get<1>(x));
    });
    float AbsoluteMeanError = AbsoluteErrorSum / Config::Data->size();
    OutputLogger::IncrementEvaluations();
    return 100 / (AbsoluteMeanError + 1);
}

shared_ptr<Expression> Expression::Initialize(shared_ptr<Expression> self,
shared_ptr<Expression> parent)
{
    if (parent != nullptr)
        self->m_parent = parent;
    for_each(self->m_subexpressions.begin(), self->m_subexpressions.end(), [&](const
shared_ptr<Expression> &subexp) {
        subexp->Initialize(subexp, self);
    });
    return self;
}
```

```
static void CountSubexpressionTypes(const shared_ptr<Expression> &e, vector<int>
&results)
{
    results.clear();
    for (int i = 0; i < 8; i++)
    {
        results.push_back(0);
    }
    auto collapsed = e->Collapse(e);
    for_each(collapsed->begin(), collapsed->end(), [&](auto subexp) {
        if (EXPRESSION_TYPE(subexp) == CONSTANT_T)
        {
            results[0] += 1;
        }
        else if (EXPRESSION_TYPE(subexp) == VARIABLE_T)
        {
            results[1] += 1;
        }
        else if (EXPRESSION_TYPE(subexp) == SIN_T)
        {
            results[2] += 1;
        }
        else if (EXPRESSION_TYPE(subexp) == COS_T)
        {
            results[3] += 1;
        }
        else if (EXPRESSION_TYPE(subexp) == PLUS_T)
        {
            results[4] += 1;
        }
        else if (EXPRESSION_TYPE(subexp) == MINUS_T)
        {
            results[5] += 1;
        }
        else if (EXPRESSION_TYPE(subexp) == DIVIDE_T)
        {
            results[6] += 1;
        }
        else if (EXPRESSION_TYPE(subexp) == MULTIPLY_T)
        {
            results[7] += 1;
        }
    });
}

float Expression::Diversity(const shared_ptr<Expression> &e1, const
shared_ptr<Expression> &e2)
{
    vector<int> e1subexp, e2subexp;
    CountSubexpressionTypes(e1, e1subexp);
    CountSubexpressionTypes(e2, e2subexp);
    float diff = 0;
    for (int i = 0; i < e1subexp.size(); i++)
    {
        diff += abs(e1subexp[i] - e2subexp[i]);
    }
    return diff;
}

shared_ptr<Expression> Expression::Simplify(shared_ptr<Expression> exp)
{
    if (exp->Depth() == 1)
    {
        return exp;
    }
    while (true)
    {
        exp = Initialize(exp, nullptr);
        exp->RecalculateLevels(0);

        // 1. find all operators
        auto collapsedExp = exp->Collapse(exp);
        if (collapsedExp->size() == 1)
            return exp;

        vector<shared_ptr<Expression>> operators;
        copy_if(collapsedExp->begin(), collapsedExp->end(),
back_inserter(operators),
                [](auto e) { return e->Order() > 0; });
        if (operators.size() == 0)
            return exp;

        // 2. check for expressions that cancels out the variables
        vector<shared_ptr<Expression>>::iterator it;
        if ((it = FindFirst(operators, subexpressionsCancelOut, minusOrDivide)) !=
operators.end())
        {
            auto expToSimplify = (*it);
            auto replacementConstant = shared_ptr<Expression>(new
Constant(expToSimplify->Level(), RandomF(0.0f, 0.2f)));
            // no parent
            if (expToSimplify->Level() == 0)
            {
                return replacementConstant;
            }
            else
            {
                ReplaceExpression(expToSimplify, replacementConstant);
            }
        }
        // 3. check for expressions that evaluates to constants
        else if ((it = FindFirst(operators, all, evaluatesToConstant)) !=
operators.end())
        {
            auto expToSimplify = (*it);

            // replace with constant at same level and of the same size
            auto replacementConstant = shared_ptr<Expression>(
                new Constant(
                    (*it)->Level(),
                    expToSimplify->ToFunction()(1)));
            //no parent
            if (expToSimplify->Level() == 0)
            {
                return replacementConstant;
            }
            else
            {
                ReplaceExpression(expToSimplify, replacementConstant);
            }
        }
        // 4. check for nested trig functions
        else if ((it = FindFirst(operators, isTrigFunction, [](auto subexp) {
                      return AnySubExpression(subexp, isTrigFunction);
                  })) != operators.end())
        {
            auto expToSimplify = (*it);

            // replace with constant at same level and of the same size
            auto randConstExpression = shared_ptr<Expression>(
                new Constant(expToSimplify->Level()));
            auto variableExprssion = shared_ptr<Expression>(
                new Variable(expToSimplify->Level()));
            auto replacementExp = shared_ptr<Expression>(
                new Multiply((*it)->Level(), randConstExpression,
variableExprssion));
            //no parent
            if (expToSimplify->Level() == 0)
            {
                return replacementExp;
            }
            else
            {
                ReplaceExpression(expToSimplify, replacementExp);
            }
        }
        else
        {
            return exp;
        }
```

```
    }
} // namespace SymbolicRegression

bool Expression::IsValid(shared_ptr<Expression> exp)
{
    auto f = exp->ToFunction();
    if (isnan(f(0)))
        return false;
    else if (EXPRESSION_TYPE(exp) == CONSTANT_T)
        return false;
    else if (exp->Depth() > Config::GetInt("MaxDepth"))
        return false;
    return true;
}

shared_ptr<vector<shared_ptr<Expression>>> Expression::Collapse(const
shared_ptr<Expression> &self) const
{
    shared_ptr<vector<shared_ptr<Expression>>> output(new
vector<shared_ptr<Expression>>());
    output->push_back(self);
    for (int i = 0; i < m_subexpressions.size(); i++)
    {
        auto collapsedSubExp = m_subexpressions[i]->Collapse(m_subexpressions[i]);
        output->insert(output->end(), collapsedSubExp->begin(), collapsedSubExp-
>end());
    }
    return output;
}

shared_ptr<Expression> Expression::GenerateRandomZeroOrderExpression(int level)
{
    if (RandomF() > 0.5f)
    {
        return shared_ptr<Expression>(new Variable(level));
    }
    else
    {
        return shared_ptr<Expression>(new Constant(level));
    }
}

shared_ptr<Expression> Expression::GenerateRandomBinaryOperator(int level)
{
    float p = RandomF();
    //equal probabilty of binary operators
    if (p > (3.0f / 4.0f))
    {
        return shared_ptr<Expression>(new Plus(level));
    }
    else if (p > (2.0f / 4.0f))
    {
        return shared_ptr<Expression>(new Minus(level));
    }
    else if (p > (1.0f / 4.0f))
    {
        return shared_ptr<Expression>(new Multiply(level));
    }
    else
    {
        return shared_ptr<Expression>(new Divide(level));
    }
}

shared_ptr<Expression> Expression::GenerateRandomTrigExpression(int level)
{
    // equal probability of cos and sin
    if (RandomF() > 0.5f)
    {
        return shared_ptr<Expression>(new Cos(level));
    }
    else
    {
        return shared_ptr<Expression>(new Sin(level));
    }
}

shared_ptr<Expression> Expression::GenerateRandomExpression(int level, bool
noConstant, bool noZero, bool noTrig)
{
    shared_ptr<Expression> exp;

    if ((RandomF() > 0.5f || level >= Config::GetInt("MaxDepth") - 1) && level > 0)
    {
        // prioritize constants
        exp = GenerateRandomZeroOrderExpression(level);
    }
    else if (RandomF() > 0.8f && !noZero && !noTrig)
    {
        //trig functions with low probability
        exp = GenerateRandomTrigExpression(level);
    }
    else
    {
        exp = GenerateRandomBinaryOperator(level);
    }
    return Simplify(exp);
}

#define pointer_cast(T, U, p) shared_ptr<T>(new U(*dynamic_pointer_cast<U>(p)));
```

```
shared_ptr<Expression> Expression::Copy(const shared_ptr<Expression> &source)
{
    shared_ptr<Expression> exp;
    string typeName(typeid(*source).name());
    string prefix("class SymbolicRegression::");
    if (typeName == prefix + "Constant")
    {
        exp = pointer_cast(Expression, Constant, source);
    }
    else if (typeName == prefix + "Variable")
    {
        exp = pointer_cast(Expression, Variable, source);
    }
    else if (typeName == prefix + "Plus")
    {
        exp = pointer_cast(Expression, Plus, source);
    }
    else if (typeName == prefix + "Minus")
    {
        exp = pointer_cast(Expression, Minus, source);
    }
    else if (typeName == prefix + "Multiply")
    {
        exp = pointer_cast(Expression, Multiply, source);
    }
    else if (typeName == prefix + "Divide")
    {
        exp = pointer_cast(Expression, Divide, source);
    }
    else if (typeName == prefix + "Cos")
    {
        exp = pointer_cast(Expression, Cos, source);
    }
    else if (typeName == prefix + "Sin")
    {
        exp = pointer_cast(Expression, Sin, source);
    }
    else
    {
        cout << "Error: Invalid type name " << typeName << endl;
    }
    exp = Initialize(exp, nullptr);
    return exp;
}

function<float(float)> Expression::ToFunction() const
{
    return m_func;
}
```

```
float Expression::RandomF()
{
    return RandomF(0, 1);
}

float Expression::RandomF(float min, float max)
{
    lock_guard<mutex> lock(randMutex);
    return (static_cast<float>(rand()) / static_cast<float>(RAND_MAX)) * (max - min)
+ min;
}

int Expression::Depth() const
{
    if (m_order == 0)
        return 1;
    int childDepth = -1;
    for (int i = 0; i < m_subexpressions.size(); i++)
    {
        childDepth = max(childDepth, m_subexpressions[i]->Depth());
    }
    return 1 + childDepth;
}

} // namespace SymbolicRegression
```

Constant.hpp

```
#ifndef _CONSTANT_HPP_
#define _CONSTANT_HPP_
#include "Expression.hpp"
#include <string>
#include <iomanip>
#include <sstream>
namespace SymbolicRegression
{
class Constant : public Expression
{
public:
    friend class ConstantMultiplierMutator;
    inline Constant(int level) : Expression(level)
    {
        m_k = SymbolicRegression::Expression::RandomF(-10, 10);
        m_func = [&](float x) { return m_k; };
        m_order = 0;
    }
    inline Constant(int level, float k) : Expression(level)
    {
        m_k = k;
        m_func = [&](float x) { return m_k; };
        m_order = 0;
    }
    inline Constant(const Constant &other) : Expression(other)
    {
        m_k = other.m_k;
        m_func = [&](float x) { return m_k; };
        m_order = 0;
    };
    inline virtual std::string ToString() const override
    {
        stringstream s;
        s << fixed << setprecision(3) << m_k;
        return s.str();
    }
    friend class ConstantMutator;

protected:
    float m_k; // value of constant
};
} // namespace SymbolicRegression
#endif
```

Divide.hpp

```
#ifndef _DIVIDE_HPP_
#define _DIVIDE_HPP_
#include "Expression.hpp"
namespace SymbolicRegression
{
class Divide : public Expression
{
public:
    inline Divide(int level) : Expression(level)
    {
        if (level + 1 == Config::GetInt("MaxDepth"))
        {

m_subexpressions.push_back(Expression::GenerateRandomZeroOrderExpression(level +
1));

m_subexpressions.push_back(Expression::GenerateRandomZeroOrderExpression(level +
1));
        }
        else
        {
            m_subexpressions.push_back(std::shared_ptr<Expression>
(Expression::GenerateRandomExpression(level + 1)));
            m_subexpressions.push_back(std::shared_ptr<Expression>
(Expression::GenerateRandomExpression(level + 1, false, true)));
        }
        m_func = [&](float x) {
            return m_subexpressions[0]->ToFunction()(x) / m_subexpressions[1]-
>ToFunction()(x);
        };
        m_order = 2;
    }

    inline Divide(const Divide &other) : Expression(other)
    {
        m_func = [&](float x) {
            return m_subexpressions[0]->ToFunction()(x) / m_subexpressions[1]-
>ToFunction()(x);
        };
        m_order = 2;
    }
    inline virtual std::string ToString() const override
    {
        return m_subexpressions[0]->ToString() + " / " + m_subexpressions[1]-
>ToString();
    }
};
} // namespace SymbolicRegression
#endif
```

Minus.hpp

```
#ifndef _MINUS_HPP_
#define _MINUS_HPP_
#include "Expression.hpp"
namespace SymbolicRegression
{
class Minus : public Expression
{
public:
    inline Minus(int level) : Expression(level)
    {
        if (level + 1 == Config::GetInt("MaxDepth"))
        {

m_subexpressions.push_back(Expression::GenerateRandomZeroOrderExpression(level +
1));
m_subexpressions.push_back(Expression::GenerateRandomZeroOrderExpression(level +
1));
        }
        else
        {
            m_subexpressions.push_back(std::shared_ptr<Expression>
(Expression::GenerateRandomExpression(level + 1)));
            m_subexpressions.push_back(std::shared_ptr<Expression>
(Expression::GenerateRandomExpression(level + 1)));
        }
        m_func = [&](float x) {
            return m_subexpressions[0]->ToFunction()(x) - m_subexpressions[1]-
>ToFunction()(x);
        };
        m_order = 2;
    }
    inline Minus(const Minus &other) : Expression(other)
    {
        m_func = [&](float x) {
            return m_subexpressions[0]->ToFunction()(x) - m_subexpressions[1]-
>ToFunction()(x);
        };
        m_order = 2;
    };
    inline std::string ToString() const override
    {
        std::string tmp = m_subexpressions[1]->ToString();
        if (m_subexpressions[1]->ToString().at(0) == '-')
            return "(" + m_subexpressions[0]->ToString() + " + " + tmp.substr(1,
tmp.length()) + ")";
        return "(" + m_subexpressions[0]->ToString() + " - " + tmp + ")";
    }
};
} // namespace SymbolicRegression
#endif
```

Plus.hpp

```
#ifndef _PLUS_HPP_
#define _PLUS_HPP_
#include "Expression.hpp"
namespace SymbolicRegression
{
using namespace std;
class Plus : public Expression
{
public:
    inline Plus(int level) : Expression(level)
    {
        if (level + 1 == Config::GetInt("MaxDepth"))
        {

m_subexpressions.push_back(Expression::GenerateRandomZeroOrderExpression(level +
1));

m_subexpressions.push_back(Expression::GenerateRandomZeroOrderExpression(level +
1));
        }
        else
        {
            m_subexpressions.push_back(std::shared_ptr<Expression>
(Expression::GenerateRandomExpression(level + 1)));
            m_subexpressions.push_back(std::shared_ptr<Expression>
(Expression::GenerateRandomExpression(level + 1)));
        }
        m_func = [&](float x) {
            return m_subexpressions[0]->ToFunction()(x) + m_subexpressions[1]-
>ToFunction()(x);
        };
        m_order = 2;
    }

    inline Plus(int level, shared_ptr<Expression> e1, shared_ptr<Expression> e2) :
Expression(level)
    {
        m_subexpressions.push_back(e1);
        m_subexpressions.push_back(e2);

        m_func = [&](float x) {
            return m_subexpressions[0]->ToFunction()(x) + m_subexpressions[1]-
>ToFunction()(x);
        };
        m_order = 2;
    }

    inline Plus(const Plus &other) : Expression(other)
    {
        m_func = [&](float x) {
            return m_subexpressions[0]->ToFunction()(x) + m_subexpressions[1]-
>ToFunction()(x);
        };
        m_order = 2;
    };
    inline virtual string ToString() const override
    {
        string tmp = m_subexpressions[1]->ToString();
        if (m_subexpressions[1]->ToString().at(0) == '-')
```

```
            return "(" + m_subexpressions[0]->ToString() + " - " + tmp.substr(1,
tmp.length()) + ")";
        return "(" + m_subexpressions[0]->ToString() + " + " + m_subexpressions[1]-
>ToString() + ")";
    }
};
} // namespace SymbolicRegression
#endif
```

Multiply.hpp

```
#ifndef _MULTIPLY_HPP_
#define _MULTIPLY_HPP_
#include "Expression.hpp"
#include "../engine/Config.hpp"

namespace SymbolicRegression
{
using namespace std;
class Multiply : public Expression
{
public:
    inline Multiply(int level) : Expression(level)
    {
        if (level + 1 == Config::GetInt("MaxDepth"))
        {

m_subexpressions.push_back(Expression::GenerateRandomZeroOrderExpression(level +
1));

m_subexpressions.push_back(Expression::GenerateRandomZeroOrderExpression(level +
1));
        }
        else
        {
            m_subexpressions.push_back(std::shared_ptr<Expression>
(Expression::GenerateRandomExpression(level + 1)));
            m_subexpressions.push_back(std::shared_ptr<Expression>
(Expression::GenerateRandomExpression(level + 1)));
        }
        m_func = [&](float x) {
            return m_subexpressions[0]->ToFunction()(x) * m_subexpressions[1]-
>ToFunction()(x);
        };
        m_order = 2;
    }

    inline Multiply(int level, shared_ptr<Expression> e1, shared_ptr<Expression> e2)
: Expression(level)
    {

        m_subexpressions.push_back(e1);
        m_subexpressions.push_back(e2);
        m_func = [&](float x) {
            return m_subexpressions[0]->ToFunction()(x) * m_subexpressions[1]-
>ToFunction()(x);
        };
        m_order = 2;
    }

    inline Multiply(const Multiply &other) : Expression(other)
    {
        m_func = [&](float x) {
            return m_subexpressions[0]->ToFunction()(x) * m_subexpressions[1]-
>ToFunction()(x);
        };
        m_order = 2;
    }

    inline virtual string ToString() const override
    {
        return m_subexpressions[0]->ToString() + " * " + m_subexpressions[1]-
>ToString();
    }
};
} // namespace SymbolicRegression
#endif
```

Sin.hpp

```
#ifndef _SIN_HPP_
#define _SIN_HPP_
#include "Expression.hpp"
namespace SymbolicRegression
{
class Sin : public Expression
{
public:
    inline Sin(int level) : Expression(level)
    {
        if (level + 1 == Config::GetInt("MaxDepth"))

m_subexpressions.push_back(Expression::GenerateRandomZeroOrderExpression(level +
1));
        else
            m_subexpressions.push_back(Expression::GenerateRandomExpression(level +
1, true, false, true));
        m_func = [&](float x) {
            return sinf(m_subexpressions[0]->ToFunction()(x));
        };
        m_order = 1;
    }

    inline Sin(int level, shared_ptr<Expression> e) : Expression(level)
    {
        m_subexpressions.push_back(e);
        m_func = [&](float x) {
            return sinf(m_subexpressions[0]->ToFunction()(x));
        };
        m_order = 1;
    }

    inline Sin(const Sin &other) : Expression(other)
    {
        m_func = [&](float x) {
            return sinf(m_subexpressions[0]->ToFunction()(x));
        };
        m_order = 1;
    };

    inline virtual std::string ToString() const override
    {
        return "sin(" + m_subexpressions[0]->ToString() + ")";
    }
};
} // namespace SymbolicRegression
#endif
```

Cos.hpp

```
#ifndef _COS_HPP_
#define _COS_HPP_
#include "Expression.hpp"
#include <math.h>
#include "../engine/Config.hpp"
namespace SymbolicRegression
{
class Cos : public Expression
{
public:
    inline Cos(int level) : Expression(level)
    {
        if (level + 1 == Config::GetInt("MaxDepth"))
m_subexpressions.push_back(Expression::GenerateRandomZeroOrderExpression(level +
1));
        else
            m_subexpressions.push_back(Expression::GenerateRandomExpression(level +
1, true, false, true));
        m_func = [&](float x) {
            return cosf(m_subexpressions[0]->ToFunction()(x));
        };
        m_order = 1;
    }

    inline Cos(const Cos &other) : Expression(other)
    {
        m_func = [&](float x) {
            return cosf(m_subexpressions[0]->ToFunction()(x));
        };
        m_order = 1;
    }

    inline virtual std::string ToString() const override
    {
        return "cos(" + m_subexpressions[0]->ToString() + ")";
    }
};
} // namespace SymbolicRegression
#endif
```

Variable.hpp

```
#ifndef _VARIABLE_HPP_
#define _VARIABLE_HPP_
#include "Expression.hpp"
namespace SymbolicRegression
{
class Variable : public Expression
{
public:
    inline Variable(int level) : Expression(level)
    {
        m_func = [&](float x) {
            return x;
        };
        m_order = 0;
        m_subexpressions.clear();
    }

    inline Variable(const Variable &other) : Expression(other)
    {
        m_func = [&](float x) {
            return x;
        };
        m_order = 0;
        m_subexpressions.clear();
    }

    inline virtual std::string ToString() const override
    {
        return "x";
    }
};
} // namespace SymbolicRegression
#endif
```