# Symbolic Regression using Genetic Programming

Name: Song WANG
UNI: sw3130

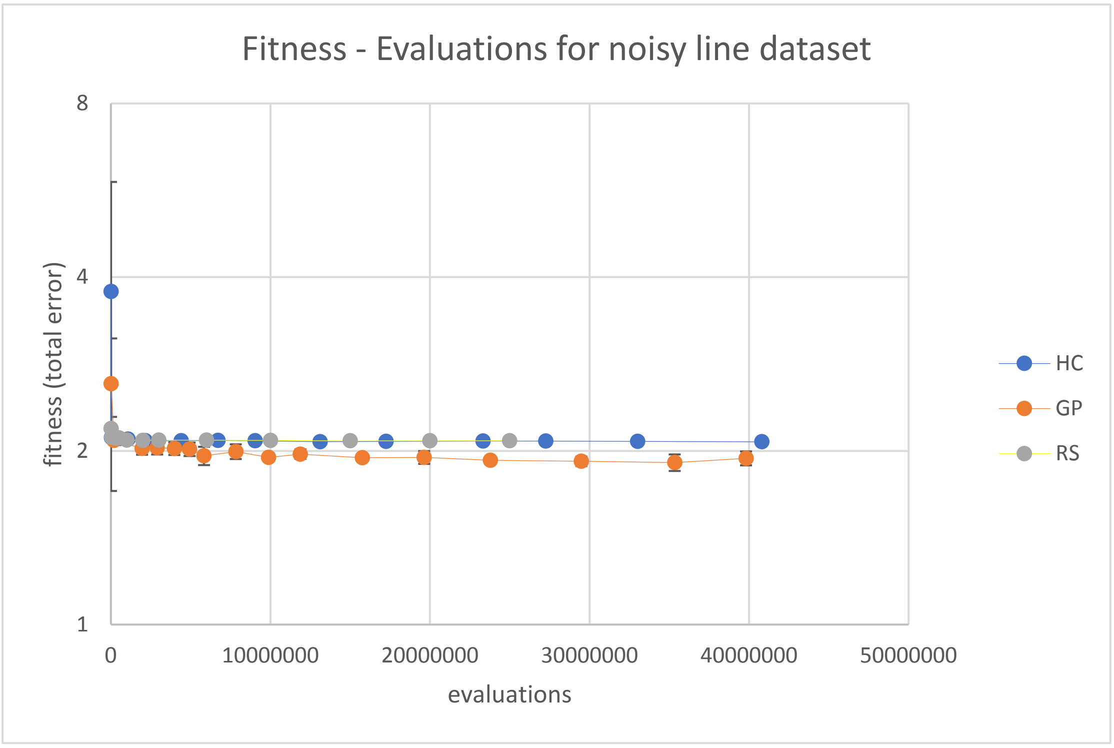
## 1. Results

### 1.1 Result for Genetic Programming, Random Search, Hill Climber

#### 1.1.1 result for noisy line file

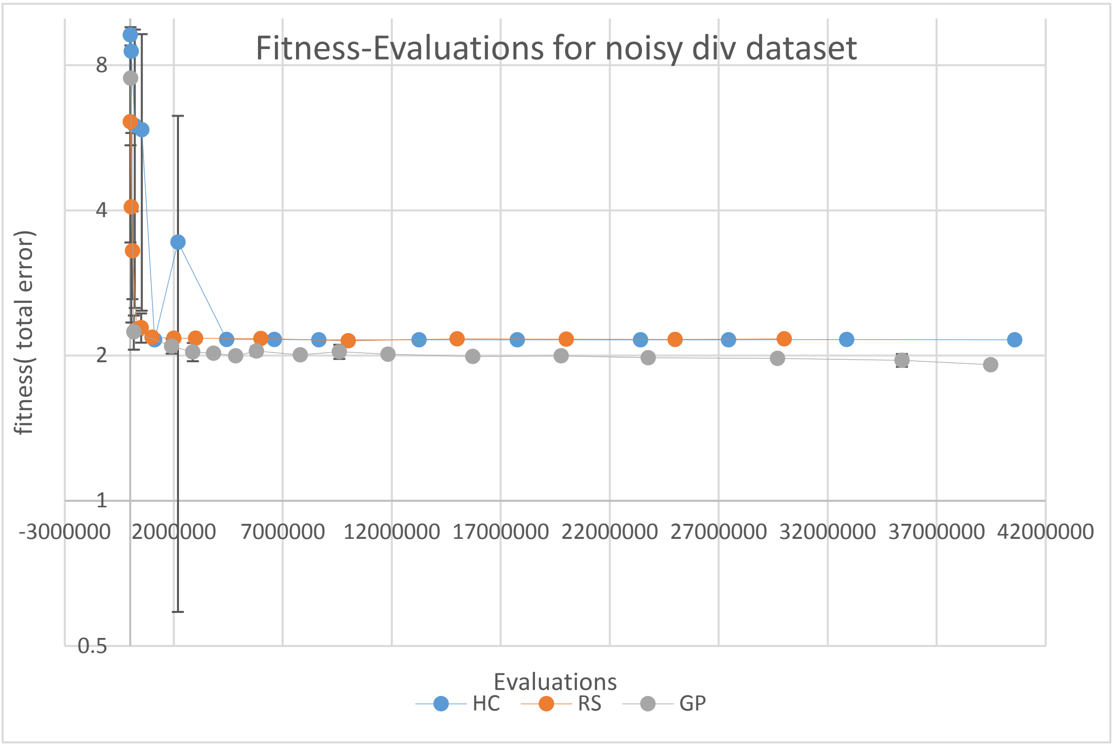
best function found:

```
f(x) = ((((((-0.11424595092035972) / x) / (-
23.564947990925248)) / ((1.9467543986915068 / x) + (-
7.847241939331861))) + ((x / 2.0738561014352355) + x))
+ (((x / 7.334805164413927) / (-29.96841824506354)) /
((3.584404879748549 / x) + (-4.29020915165005))))
least total error:1.8495994257458392
number of evaluations:  39811567
Running time: 150446ms
```



1.1.2 result for noisy div file

```
best function found:
f(x) = ((((-0.2009763919755354) / (42.26650397967963 +
(5.565273040197489 / x))) + x) + ((x / ((-
0.00200900645087998) + x)) / (39.7226325653638 * x)))
least total error:TotalError = 1.878769867969262
number of evaluations:  58747837
Running time:271843ms
```



## 2. Algorithm Implementation

### 1.1 Genetic Programming

#### 1.1.1 Vocabulary

A Vocab.java is created to store all the operators or variables that will be used and can be viewed as the material to be used for building the binary tree or let’s say the vocabulary for the binary tree. This class was mainly composed of getters and setters.

#### 1.1.2 Individual

If we want to optimize a binary tree representation of a function, we not only should treat the binary tree itself as an individual and optimize it, what’s more here I also treat the coefficients which formed a list as an coefficient individual since usually to get a better result we only need to change(mutate) the coefficients, for example if we get a linear relation for the line dataset, randomly change the node of the binary tree representation won’t help, in this case, change the coefficient is necessary.  So we have two kinds of Individual, an Individual Interface is constructed in the java project, for Genetic programming, we require each individual to mutate and do cross between two individual, so mutate method and cross method is require in the interface for implemented class from this interface.

#### 1.1.3 Population

After defining two kinds of Individual in the GPIndividual.java file, a population class is created to manage the group of individuals, this class is mainly composed of getters and setters.

#### 1.1.4 Mutation

Individuals are defined in the GPIndividual.java file, in this file mutation and cross methods are also implemented for two kinds of individual. For coefficients of a binary tree, the only mutation method is to change its’s value and doing cross for two coefficient individuals. For a binary tree, the cross method is just like an ordinary cross in genetic algorithm, but there are multiple mutation methods that I can think of (maybe these methods have some overlap).

(1) Replace the whole binary tree with its any subtree, since a tree is constructed recursively, we only need to replace the root node to any of its’s sub node.
(2) Increase the depth of the tree by one, first obtain a non-terminal operator from the vocabulary we built

since all the non-terminal operator we used are binary so just add the original tree as one of the child of the operator we choose, then randomly add another child.

(3) Change one of the children of given node (binary tree), choose a random node, due to the symmetry of the node, then always choose its first child, and set this node as a new tree.

(4) Just randomly create a new tree to replace the original subtree

1.1.5 Fitness Calculation

Here use the total absolute error for all the 50 data points as fitness. The absolute error is the absolute value of the true value minus the calculated value based on the binary tree we created.

1.1.6 Evolve

For Evolution, given the population of last generation, the best individual of last generation is added into current generation, then for each parent individual, the mutated parent and the crossed parent are added into current generation, then we sort the individual (this step also costs lots of evaluation) based on their fitness, and choose the number of individuals we set to be the current generation.

1.1.7 Driver

After everything is settled, a driver class is incorporated to receive input data, create initial population and drive the genetic algorithm to run.

1.1.8 Counter

A counter is set at the fitness calculation function to log the number of evaluations

1.2 Random Search

1.2.1 In order to reuse the code written for Genetic Programming, for random search, I leave the evolve method in the driver blank, no mutation and cross method is called in Random Search, then randomly create an initial population in the driver and linear search the best binary tree representation from the population

1.3 Hill Climber

1.3.1 In order to reuse the code for Genetic Programming, in the evolve method in drive class, we initially randomly create a population, then linearly select the best tree representation, mutate the best of given times and add them all into a new generation (include the original best tree), select the best tree into the next hill climb step, also no cross method is called here.

3. Code list

1.1 Summary of the code

The hierarchies was generated by the default java development kit (version 1.8.0_131) Javadoc tool.

## Hierarchy For All Packages

**Package Hierarchies:**

- SRGP

### Class Hierarchy

- java.lang.Object
  - SRGP.**Driver**
  - SRGP.**Evolve**<I,T>
  - SRGP.**Main_GP**
  - SRGP.**Node** (implements java.lang.Cloneable)
  - SRGP.**Population**<I>
  - SRGP.**ReadData**
  - SRGP.**Vocab**

### Interface Hierarchy

- SRGP.**Fitness**<I,T>
- SRGP.**Individual**<I>
- SRGP.**Operator**

### Enum Hierarchy

- java.lang.Object
  - java.lang.Enum<E> (implements java.lang.Comparable<T>, java.io.Serializable)

- SRGP.**Operators** (implements SRGP.Operator)

File link (html file) : this

### 1.2 List of Java file

Java file link and the documentations : this

#### 1.2.1 Driver.java

```java
package SRGP;

import java.util.Random;

// TODO: Auto-generated Javadoc
/**
 * The Class Driver.
 */
public class Driver {

    /** The Constant POPULATION_SIZE. */
    static final int POPULATION_SIZE = 20;

    /** The evolver. */
    private Evolve<GPIndividual, Double> evolver;

    /** The fitness function. */
    private SRFitness fitnessFunction;

    /**
     * Instantiates a new driver.
     *
     * @param input the input
     */
    public Driver(Double[][] input) {
        fitnessFunction = new SRFitness(input);
        Population<GPIndividual> population = this.createPopulation( );
        this.evolver = new Evolve<GPIndividual, Double>(population, fitnessFunction);
    }

    /**
     * initially we always create a tree with a depth of depth from 1 to 5;
     *
     * @author wangsong
     * @return the population
     */
    private Population<GPIndividual> createPopulation() {
        Random random = new Random();
        Population<GPIndividual> population = new Population<GPIndividual>();
        for (int i = 0; i < POPULATION_SIZE; i++) {
            GPIndividual individual = new GPIndividual(this.fitnessFunction,
GPIndividual.createTree(random.nextInt(5) + 1));
            population.addIndividual(individual);
        }
        return population;
    }

    /**
     * Evolve.
     *
     * @param iters the iters
     */
    public void evolve(int iters) {
        this.evolver.evolve(iters);
    }

    /**
     * Gets the best tree.
     *
     * @return the best tree
     */
    public Node getBestTree() {
        return this.evolver.getFittest().getTree();
    }

    /**
     * Fitness.
     *
     * @param n the n
     * @return the double
     */
    public double fitness(Node n) {
        return this.fitnessFunction.fitness(n);
    }
}
```

### 1.2.2 Evolve.java

```
package SRGP;

import java.util.*;

// TODO: Auto-generated Javadoc
/**
 * The Class Evolve.
 *
 * @param <I> the generic type
 * @param <T> the generic type
 */
public class Evolve<I extends Individual<I>, T extends Comparable<T>> {

    /** The comparator. */
    private final IndividualComparator comparator = new IndividualComparator();

    /** The fitness function. */
    final Fitness<I, T> fitnessFunction;

    /** The population. */
    private Population<I> population;

    /**
     * given a population and a fitness function to calculate individual fitness
     * a evovler can be implemented.
     *
     * @param population the population
     * @param fitnessF the fitness F
     */
    public Evolve(Population<I> population, Fitness<I, T> fitnessF) {
        this.population = population;
        this.fitnessFunction = fitnessF;
    }

    // for each generation we need to sort the fitness of the individual
    // for a list with a class type we need to calculate its fitness first
    /**
     * The Class IndividualComparator.
     */
```

```java
	// then sort the objects, so construct a comparator here.
	private class IndividualComparator implements Comparator<I> {

		/** The hm. */
		private final Map<I, T> hm = new HashMap<I, T>();

		/* (non-Javadoc)
		 * @see java.util.Comparator#compare(java.lang.Object, java.lang.Object)
		 */
		@Override
		public int compare(I individual1, I individual2) {
			T fit1 = this.fitness(individual1);
			T fit2 = this.fitness(individual2);
			return fit1.compareTo(fit2);
		}

		/**
		 * Fitness.
		 *
		 * @param individual the individual
		 * @return the t
		 */
		public T fitness(I individual) {
			T fit = this.hm.get(individual);
			if (fit == null) {
				fit = Evolve.this.fitnessFunction.calculate(individual);
				this.hm.put(individual, fit);
			}
			return fit;
		};
	}

	/**
	 * Gets the fittest.
	 *
	 * @return the fittest
	 */
	public I getFittest() {
		I result = this.population.getIndividual(0);
		T min =
Evolve.this.fitnessFunction.calculate(this.population.getIndividual(0));
		for (int i = 0; i < this.population.getSize(); i ++) {
			T fit =
Evolve.this.fitnessFunction.calculate(this.population.getIndividual(i));
```

```
            if (fit.compareTo(min) < 0) {
                min = fit;
                result = this.population.getIndividual(i);
            }
        }
        return result;
    }

    /**
     * every generation we get the fittest parent individual, and copy it into the
next
     * generation,.
     *
     * @author wangsong
     */
    public void evolve() {
        int parentSize = this.population.getSize();
        Population<I> newPopulation = new Population<I>();
        // select the best parent and move to new population
        newPopulation.addIndividual(Evolve.this.getFittest());
        for (int i = 0; i < parentSize; i++) {
            I individual = this.population.getIndividual(i);
            I mutated = individual.mutate();
            I otherIndividual = this.population.getIndividual();
            List<I> crossed = individual.cross(otherIndividual);
            newPopulation.addIndividual(mutated);
            for (I ind : crossed) {
                newPopulation.addIndividual(ind);
            }
        }
        // this sorting evaluates the fitness function a lot
        newPopulation.sortPopulation(this.comparator);
        newPopulation.trim(parentSize);
        this.population = newPopulation;
    }

    /**
     * Evolve.
     *
     * @param count the count
     */
    public void evolve(int count) {
        for (int i = 1; i <= count; i++) {
            this.evolve();
        }
    }
}
```

### 1.2.3 Fitness.java

```
package SRGP;

// TODO: Auto-generated Javadoc
/**
 * the coefficients of a tree is treated as an individual, also the coefficients will
be evolved,
 * but the calculation between a tree and the coefficients individual is different, so
a interface
 * for calculate fitness is required .
 *
 * @author wangsong
 * @param <T> the generic type
 */
public interface Fitness<I extends Individual<I>, T extends Comparable<T>> {

    /**
     * Calculate.
     *
     * @param individual the individual
     * @return the t
     */
    T calculate(I individual);
                    }
```

### 1.2.4 GPIndividual.java

```
package SRGP;

import java.util.*;

// TODO: Auto-generated Javadoc
/**
 * The Class GPIndividual.
 */
class GPIndividual implements Individual<GPIndividual> {

    /** The random. */
    private Random random = new Random();

    /** The b tree. */
    private Node bTree;
    /** The fitness function. */
    private Fitness<GPIndividual, Double> fitnessFunction;

    /**
     * Instantiates a new GP individual.
     *
     * @param fitnessFunction the fitness function
     * @param tree the tree
     */
    public GPIndividual(Fitness<GPIndividual, Double> fitnessFunction, Node tree) {
        this.fitnessFunction = fitnessFunction;
        this.bTree = tree;
    }

    /* (non-Javadoc)
     * @see SRGP.Individual#mutate()
     */
    @Override
    public GPIndividual mutate() {
        GPIndividual result = new GPIndividual(this.fitnessFunction,
this.bTree.clone());
        int num = this.random.nextInt(4);
        if (num == 0) result.replaceSubTree();
        if (num == 1) result.changeChild();
        if (num == 2) result.increaseDepth();
        if (num == 3) result.bTree = createTree(3);
        result.optimize(20);
        return result;
    }

    /**
     * replace the whole tree with any subtree.
     */
    private void replaceSubTree() {
        this.bTree = this.getRandomNode(this.bTree);
    }

    /**
     * obtain a non-terminal operator from the vocabulary we built
     * since all the non-terminal operator we used are binary
     * so just add the original tree as one of the child of the
     * operator we choose, then randomly add another child.
     */
    private void increaseDepth() {
        Operator o = new Vocab().getOneNonTerminalOperator();
        Node newRoot = new Node(o);
        newRoot.addChild(this.bTree);
        newRoot.addChild(createTree(0));
        newRoot.addCoeff(random.nextDouble() * 20 - 10);
        this.bTree = newRoot;
    }

    /**
     * choose a random node, then always choose its first child,
     * and set this node as a new tree,if the node has no child at all,
     * then change the operator of the node.
     */
    private void changeChild() {
        Node node = this.getRandomNode(this.bTree);
        if (!node.getChildren().isEmpty()) {
            node.getChildren().set(0, createTree(1));
        }
    }

    /* (non-Javadoc)
     * @see SRGP.Individual#cross(SRGP.Individual)
     */
    @Override
    public List<GPIndividual> cross(GPIndividual aIndividual) {
        ArrayList<GPIndividual> result = new ArrayList<GPIndividual>(2);
        GPIndividual aClone = new GPIndividual(this.fitnessFunction,
this.bTree.clone());
        GPIndividual bClone = new GPIndividual(aIndividual.fitnessFunction,
aIndividual.bTree.clone());

    this.swapNode(this.getRandomNode(aClone.bTree),aIndividual.getRandomNode(bClone.bT
ree).clone());
        this.swapNode(aIndividual.getRandomNode(bClone.bTree),
this.getRandomNode(aClone.bTree).clone());
        aClone.optimize(20);
        result.add(aClone);
        bClone.optimize(20);
        result.add(bClone);
        return result;
    }

    /**
     * get a random node from all the nodes that are in a
     * subtree with a root as Node tree.
     *
     * @param tree the tree
     * @return a random node
     */
    private Node getRandomNode(Node tree) {
        ArrayList<Node> allNodesOfTree = tree.getAllNodes();
        return allNodesOfTree.get(this.random.nextInt(allNodesOfTree.size()));
    }

    /**
     * in a tree, change a node to a new node(to be a root of
     * of a new subtree) can simply implemented by change the
     * operator of this node, and set its children to the new
     * node's children.
     *
     * @param oldNode the old node
     * @param newNode the new node
     */
    private void swapNode(Node oldNode,Node newNode) {
        oldNode.setChildren(newNode.getChildren());
        oldNode.setOperator(newNode.getOperator());
        oldNode.setCoeffOfNode(newNode.getCoeffOfNode());
    }

    /**
     * for simplify or optimize the tree, we first cut the tree to be a depth of
     * at most 5; then simply the tree by replacing all the subtree who is made up of
     * only constants with the evaluated value of the subtree;.
     *
     * @param iterations the iterations
     */
    public void optimize(int iterations) {
        cutTree(this.bTree, 5);
        simplifyTree(this.bTree);
        List<Double> coeffOfTree = this.bTree.getCoeffOfTree();

        if (coeffOfTree.size() > 0) {
            CoeffIndividual initialIndividual = new CoeffIndividual(coeffOfTree);
            Population<CoeffIndividual> population = new
Population<CoeffIndividual>();
            for (int i = 0; i < 3; i++) {
                population.addIndividual(initialIndividual.mutate());
            }
```

```
                population.addIndividual(initialIndividual);
                Fitness<CoeffIndividual, Double> fit = new CoeffFitness();
                Evolve<CoeffIndividual, Double> evolver = new Evolve<CoeffIndividual,
Double>(population, fit);
                evolver.evolve(iterations);
                List<Double> optimizedCoeff = evolver.getFittest().getCoeff();
                this.bTree.setCoeffOfTree(optimizedCoeff);
            }
        }

        /**
         * Gets the tree.
         *
         * @return the tree
         */
        public Node getTree() {
            return this.bTree;
        }

        /**
         * The Class CoeffIndividual.
         */
        private class CoeffIndividual implements Individual<CoeffIndividual>, Cloneable {
            /** The coeff. */
            private List<Double> coeff;

            /**
             * just create a individual where the coefficients is a tree.
             *
             * @param coeff the coeff
             */
            public CoeffIndividual(List<Double> coeff) {
                this.coeff = coeff;
            }

            /**
             * construct two Coefficient individuals which are coefficient of constants
of a subtree
             * make two backup of these two coeffIndividuals.
             *
             * @param bIndividual the b individual
             * @return the list
             */
            @Override
```

```java
public List<CoeffIndividual> cross(CoeffIndividual bIndividual) {
	List<CoeffIndividual> result = new ArrayList<CoeffIndividual>(2);

	CoeffIndividual aClone = this.clone();
	CoeffIndividual bClone =bIndividual.clone();

	for (int i = 0; i < aClone.coeff.size(); i++) {
		if ( Math.random() > 0.3) {
			aClone.coeff.set(i, bIndividual.coeff.get(i));
			bClone.coeff.set(i, this.coeff.get(i));
		}
	}
	result.add(aClone);
	result.add(bClone);

	return result;
}

/* (non-Javadoc)
 * @see SRGP.Individual#mutate()
 */
@Override
public CoeffIndividual mutate() {
	CoeffIndividual result = this.clone();
	for (int i = 0; i < result.coeff.size(); i++) {
		if (Math.random() > 0.5) {
			double coeff = result.coeff.get(i);
			coeff = coeff + Math.random() * 10 - 5;
			result.coeff.set(i, coeff);
		}
	}
	return result;
}

/* (non-Javadoc)
 * @see java.lang.Object#clone()
 */
@Override
protected CoeffIndividual clone() {
	List<Double> result = new ArrayList<Double>(this.coeff.size());
	for (double d : this.coeff) {
		result.add(d);
	}
	return new CoeffIndividual(result);
```

```
        }

        /**
         * Gets the coeff.
         *
         * @return the coeff
         */
        public List<Double> getCoeff() {
            return this.coeff;
        }

    }

    /**
     * The Class CoeffFitness.
     *
     * @author wangsong
     */
    private class CoeffFitness implements Fitness<CoeffIndividual, Double> {

        /* (non-Javadoc)
         * @see SRGP.Fitness#calculate(SRGP.Individual)
         */
        @Override
        public Double calculate(CoeffIndividual individual) {
            GPIndividual.this.bTree.setCoeffOfTree(individual.getCoeff());
            return GPIndividual.this.fitnessFunction.calculate(GPIndividual.this);
        }

    }

    /**
     * Creates the tree.
     *
     * @param depth the depth
     * @return the node
     */
    public static Node createTree(int depth) {
        Vocab vocab = new Vocab();
        if (depth > 0) {
            Operator o;
            if (Math.random() >= 0.3) {
                o = vocab.getOneNonTerminalOperator();
            } else {
```

```java
            o = vocab.getOneTerminalOperator();
        }
        Node n = new Node(o);
        if (o.numArg() > 0) {
            for (int i = 0; i < o.numArg(); i++) {
                Node child = createTree(depth - 1);
                n.addChild(child);
            }
        }
        if (o.isConstant()) {
            n.addCoeff(Math.random() * 20 - 10);
        }
        return n;
    } else {
        Operator o = vocab.getOneTerminalOperator();
        Node n = new Node(o);
        if (o.isConstant()) {
            n.addCoeff(Math.random() * 20 - 10);
        }
        return n;
    }
}

/**
 * as the tree continue to grow, the tree gets very messy
 * we create this method to simply the tree; usually if the
 * subtree is completely made up with constant, just replace
 * the subtree to a single constant node whose value is equal
 * to the subtree evaluated under the vocabulary.
 *
 * @author wangsong
 * @param tree the root of the subtree to be simplified
 */
public static void simplifyTree(Node tree) {
    Vocab vocab = new Vocab();
    if (Node.hasVariableNode(tree)) {
        for (Node child : tree.getChildren()) {
            simplifyTree(child);
        }
    } else {
        double value = tree.evaluate(vocab);
        tree.addCoeff(value);
        tree.clear();
        tree.setOperator(vocab.getOperators().get(5));
        }
    }

    /**
     * given a tree node to cut the subtree rooted at this node to
     * a subtree with depth of at most 5.
     *
     * @author wangsong
     * @param tree the tree
     * @param depth the depth
     */
    public static void cutTree(Node tree,int depth) {
        Vocab vocab = new Vocab();
        if (depth > 0) {
            for (Node child : tree.getChildren()) {
                cutTree(child, depth - 1);
            }
        } else {
            tree.clear();
            tree.clearCoeff();
            Operator o = vocab.getOneTerminalOperator();
            tree.setOperator(o);
            if (o.isConstant()) {
                tree.addCoeff(Math.random() * 20 - 10);
            }
        }
    }
}
```

### 1.2.5 Individual.java

```java
package SRGP;

import java.util.List;

// TODO: Auto-generated Javadoc
/**
 * the coefficients of a tree is treated as an individual, also the coefficients will
be evolved,
 * so a interface for individual is required.
 *
 * @author wangsong
 * @param <I> the generic type
 */
public interface Individual<I extends Individual<I>> {
    /**
     * Cross.
     *
     * @param individual the individual
     * @return the list
     */
    List<I> cross( I individual );

    /**
     * Mutate.
     *
     * @return the i
     */
    I mutate();
}
```

1.2.6   Main_GP.java

```
package SRGP;

// TODO: Auto-generated Javadoc
/**
 * The Class Main_GP.
 */
public class Main_GP {

    /**
     * The main method.
     *
     * @param args the arguments
     */
    public static void main(String[] args) {

        //String fileName = "C:/Users/wangsong/Documents/Symbolic 
Regression/SR_line.txt";
        String fileName = "C:/Users/wangsong/Documents/Symbolic 
Regression/SR_div_noise.txt";
        Double[][] input = new ReadData(fileName).read();
    Driver driver = new Driver(input);
        driver.evolve(20);
        double fitness = driver.fitness(driver.getBestTree());
        System.out.println(String.format("TotalError = %s   f(x) = %s", fitness, 
driver.getBestTree().print()));
        System.out.println("number of evaluations:  " + SRFitness.count);
    }
}
```

### 1.2.7 Node.java

```java
package SRGP;

import java.util.*;

// TODO: Auto-generated Javadoc
/**
 * after the vocabulary, I need to create this class to act as the
 * basic arithmetic unit which deal with a binary computation in our
 * case (add, subtract, multiply, divide), and for a single node, its
 * children (left or right can be a terminal or expressions themselves).
 * In case for the tree can mutate, crossover (which means a node's
 * children can be changed, so we need to create multiple methods to
 * get this nodes's children and add a subtree as this node's child.
 * What's more in order to evaluate a Tree, we can do it in a recursive
 * way, first evaluate it's left subtree, then it's right subtree, lastly
 * combined with the node's operator to do the computation.
 * here I just call it node here, and the node itself or the subtree start
 * from the node are all called node here since it's a standard class for a tree node
 * @author wangsong
 *
 */

public class Node implements Cloneable {

    /** The children. */
    private ArrayList<Node> children = new ArrayList<Node>();

    /** The coeff. */
    // will optimize the coefficients of the tree
    private ArrayList<Double> coeff = new ArrayList<Double>();

    /** The o. */
    private Operator o;

    /**
     * Instantiates a new node.
     *
     * @param o the o
     */
    public Node (Operator o) {
        this.o = o;
    }
```

```java
/**
 * Evaluate.
 *
 * @param vocab the vocab
 * @return the double
 */
public double evaluate(Vocab vocab) {
     return this.o.evaluate(this, vocab);
}

/**
 * Prints the.
 *
 * @return the string
 */
public String print() {
     return this.o.print(this);
}

/**
 * Gets the children.
 *
 * @return the children
 */
// getters and setters for instance variables
public ArrayList<Node> getChildren() {
     return this.children;
}

/**
 * Sets the children.
 *
 * @param children the children
 * @return the node
 */
public Node setChildren(ArrayList<Node> children) {
     this.children = children;
     return this;
}

/**
 * Adds the child.
 *
 * @param child the child
 */
public void addChild(Node child) {
     this.children.add(child);
}

/**
 * Clear.
 */
public void clear() {
     this.children.clear();
}

/**
 * Gets the coeff of node.
 *
 * @return the coeff of node
 */
public ArrayList<Double> getCoeffOfNode() {
     return this.coeff;
}

/**
 * Sets the coeff of node.
 *
 * @param coeff the coeff
 * @return the node
 */
public Node setCoeffOfNode(ArrayList<Double> coeff) {
     this.coeff = coeff;
     return this;
}

/**
 * Adds the coeff.
 *
 * @param coeff the coeff
 */
public void addCoeff(double coeff) {
     this.coeff.add(coeff);
}

/**
 * Clear coeff.
 */
```

```
public void clearCoeff() {
    if (this.coeff.size() > 0) {
        this.coeff.clear();
    }
}

/**
 * Gets the operator.
 *
 * @return the operator
 */
public Operator getOperator() {
    return this.o;
}

/**
 * Sets the operator.
 *
 * @param operator the new operator
 */
public void setOperator(Operator operator) {
    this.o = operator;
}

/* (non-Javadoc)
 * @see java.lang.Object#clone()
 */
@Override
public Node clone() {
    Node cloned = new Node(this.o);

    for (Node c : this.children) {
        cloned.children.add(c.clone());
    }
    for (Double d : this.coeff) {
        cloned.coeff.add(d);
    }
    return cloned;
}

/**
 * Gets the coeff of tree.
 *
 * @return the coeff of tree
 */
public List<Double> getCoeffOfTree() {
    LinkedList<Double> coeff = new LinkedList<Double>();
    this.getCoeffOfTree(coeff);
    Collections.reverse(coeff);
    return coeff;
}

/**
 * Gets the coeff of tree.
 *
 * @param coeff the coeff
 * @return the coeff of tree
 */
private void getCoeffOfTree(Deque<Double> coeff) {
    List<Double> coeffs = this.o.getCoeff(this);
    for (Double d : coeffs) {
        coeff.push(d);
    }
    for (int i = 0; i < this.children.size(); i++) {
        this.children.get(i).getCoeffOfTree(coeff);
    }
}

/**
 * Sets the coeff of tree.
 *
 * @param coeff the new coeff of tree
 */
public void setCoeffOfTree(List<Double> coeff) {
    this.setCoeffOfTree(coeff, 0);
}

/**
 * Sets the coeff of tree.
 *
 * @param coeff the coeff
 * @param index the index
 * @return the int
 */
private int setCoeffOfTree(List<Double> coeff, int index) {
    this.o.setCoeff(this, coeff, index);
    if (this.o.isConstant()) {
        index += 1;
```

```java
    }
    if (this.children.size() > 0) {
        for (int i = 0; i < this.children.size(); i++) {
            index = this.children.get(i).setCoeffOfTree(coeff, index);
        }
    }
    return index;
}

/**
 * Gets the all nodes.
 *
 * @return the all nodes
 */
public ArrayList<Node> getAllNodes() {
    ArrayList<Node> nodes = new ArrayList<Node>();
    int index = 0;
    nodes.add(this);
    while (true) {
        if (index < nodes.size()) {
            Node node = nodes.get(index++);
            for (Node child : node.children) {
                nodes.add(child);
            }
        } else {
            break;
        }
    }
    return nodes;
}

/**
 * traversal all the nodes in the subtree rooted at tree
 * and return true if any operator is a variable.
 *
 * @author wangsong
 * @param tree the tree
 * @return true if any operator in the tree is a variable
 */
public static boolean hasVariableNode(Node tree) {
    boolean result = false;
    if (!tree.getOperator().isConstant() && tree.getOperator().numArg() == 0) {
        result = true;
    } else {
            for (Node child : tree.getChildren()) {
                result = hasVariableNode(child);
                if (result) {
                    break;
                }
            }
        }
        return result;
    }

}
```

### 1.2.8 Operator.java

```java
package SRGP;

import java.util.List;

// TODO: Auto-generated Javadoc
/**
 * The Interface Operator.
 */
public interface Operator {

    /**
     *  for each operator the calculation of the tree is
     *       different, so evaluate method is required in the interface.
     *
     * @param n the n
     * @param vocab the vocab
     * @return the double
     */
    double evaluate(Node n, Vocab vocab);

    /**
     *  constants and variable node are terminal nodes, their argument
     *    number should be zero, then for nonterminal nodes
add/divide/multiply/subtract,
     *    their argument number is two, in order to differentiate terminal nodes from
     *    nonterminal, a method to return the number of arguments is required in the
interface.
     *
     * @return the int
     */
    int numArg();
    /**
     * if the operator is a constant, since I will optimize the constants of
     * the syntax tree, and make these constants to form a individual and do cross and
mutate, so
     * a method to determine whether it's a constant or not is required in the
interface.
     *
     * @return true if it's a constant
     */
    boolean isConstant();

    /**
     * since if I want to recursively print a tree, and print method is different for
different node
     * so a method to require each operator to implement should be set at the
interface .
     *
     * @param n root of the tree to be printed
     * @return the string
     */
    String print(Node n);

    /**
     * I will optimize the constants of the tree, so a method to return all the
constants in the subtree
     * rooted at node n is required in the interface.
     *
     * @param n the n
     * @return the coeff
     */
    List<Double> getCoeff(Node n);

    /**
     * after get the constants of the subtree, and doing cross and mutate for the
constants individual,
     * we need a method to set the mutated constants back into the tree, sefConst
method is required
     * in the interface.
     *
     * @param n the n
     * @param coeff the coeff
     * @param index the index
     */
    void setCoeff(Node n, List<Double> coeff, int index);
```

```
}
```

### 1.2.9 Operators.java

```
package SRGP;

import java.util.*;

// TODO: Auto-generated Javadoc
/**
 * The Enum Operators.
 */
public enum Operators implements Operator {

    /** The constant. */
    CONSTANT {
        @Override
        public String print(Node n) {
            double coeff = n.getCoeffOfNode().get(0);
            String result = null;
            if (coeff < 0) {
                result = String.format("(%s)", coeff);
            } else {
                result = "" + coeff;
            }
            return result;
        }

        @Override
        public int numArg() {
            return 0;
        }

        @Override
        public List<Double> getCoeff(Node n) {
            return n.getCoeffOfNode().subList(0, 1);
        }

        @Override
        public void setCoeff(Node n, List<Double> coeff, int index) {
            n.clearCoeff();
            n.addCoeff(coeff.get(index));
        }
```

```java
        @Override
        public double evaluate(Node n, Vocab vocab) {
            return n.getCoeffOfNode().get(0);
        }

        @Override
        public boolean isConstant() {
            return true;
        }
    },

    /** The variable. */
    VARIABLE {

        @Override
        public List<Double> getCoeff(Node n) {
            return new LinkedList<Double>();
        }

        @Override
        public void setCoeff(Node n, List<Double> coeff, int index) {
            n.clearCoeff();
        }

        @Override
        public int numArg() {
            return 0;
        }

        @Override
        public boolean isConstant() {
            return false;
        }

        @Override
        public double evaluate(Node n, Vocab vocab) {
            return vocab.getVariableValue();
        }

        @Override
        public String print(Node n) {
            return "x";
        }
```

```
},

/** The add. */
ADD {

    @Override
    public List<Double> getCoeff(Node n) {
        return new LinkedList<Double>();
    }

    @Override
    public void setCoeff(Node n, List<Double> coeff, int index) {
        n.clearCoeff();
    }

    @Override
    public int numArg() {
        return 2;
    }

    @Override
    public boolean isConstant() {
        return false;
    }



    @Override
    public double evaluate(Node n, Vocab vocab) {
        List<Node> children = n.getChildren();
        double left = children.get(0).evaluate(vocab);
        double right = children.get(1).evaluate(vocab);
        return (left + right);
    }

    @Override
    public String print(Node n) {
        List<Node> children = n.getChildren();
        String left = children.get(0).print();
        String right = children.get(1).print();
        return String.format("(%s + %s)", left, right);
    }
},

/** The sub. */
```

```java
SUB {


    @Override
    public List<Double> getCoeff(Node n) {
        return new LinkedList<Double>();
    }

    @Override
    public void setCoeff(Node n, List<Double> coeff, int index) {
        n.clearCoeff();
    }

    @Override
    public int numArg() {
        return 2;
    }

    @Override
    public boolean isConstant() {
        return false;
    }



    @Override
    public double evaluate(Node n, Vocab vocab) {
        List<Node> children = n.getChildren();
        double left = children.get(0).evaluate(vocab);
        double right = children.get(1).evaluate(vocab);
        return (left - right);
    }

    @Override
    public String print(Node n) {
        List<Node> children = n.getChildren();
        String left = children.get(0).print();
        String right = children.get(1).print();
        return String.format("(%s - %s)", left, right);
    }
},

/** The mul. */
MUL {
        @Override
        public List<Double> getCoeff(Node n) {
            return new LinkedList<Double>();
        }

        @Override
        public void setCoeff(Node n, List<Double> coeff, int index) {
            n.clearCoeff();
        }

        @Override
        public int numArg() {
            return 2;
        }

        @Override
        public boolean isConstant() {
            return false;
        }

        @Override
        public double evaluate(Node n, Vocab vocab) {
            List<Node> children = n.getChildren();
            double left = children.get(0).evaluate(vocab);
            double right = children.get(1).evaluate(vocab);
            return (left * right);
        }

        @Override
        public String print(Node n) {
            List<Node> children = n.getChildren();
            String left = children.get(0).print();
            String right = children.get(1).print();
            return String.format("(%s * %s)", left, right);
        }
    },

    /** The div. */
    DIV {

        @Override
        public List<Double> getCoeff(Node n) {
            return new LinkedList<Double>();
        }
        @Override
        public void setCoeff(Node n, List<Double> coeff, int index) {
            n.clearCoeff();
        }

        @Override
        public int numArg() {
            return 2;
        }

        @Override
        public boolean isConstant() {
            return false;
        }



        @Override
        public double evaluate(Node n, Vocab vocab) {
            List<Node> children = n.getChildren();
            double left = children.get(0).evaluate(vocab);
            double right = children.get(1).evaluate(vocab);
            return (left / right);
        }

        @Override
        public String print(Node n) {
            List<Node> children = n.getChildren();
            String left = children.get(0).print();
            String right = children.get(1).print();
            return String.format("(%s / %s)", left, right);
        }
    },

}
```

### 1.2.10 Population.java

```
package SRGP;

import java.util.*;

// TODO: Auto-generated Javadoc
/**
 * The Class Population.
 *
```

```java
 * @param <I> the generic type
 */
public class Population<I extends Individual<I>> {

    /**  usually after last generation, the population size is the constant I set
     * at the beginning of the next generation we first add the best parent to
     * current population, then each parent is mutated and each pair parents are
     * crossed, so the population size should be set at POPULATION_SIZE * 3 + 1. */
    private List<I> individuals = new ArrayList<I>(Driver.POPULATION_SIZE * 3 + 1);

    /** The random. */
    private final Random random = new Random();

    /**
     * Adds the individual.
     *
     * @param individual the individual
     */
    public void addIndividual(I individual) {
        this.individuals.add(individual);
    }

    /**
     * Gets the size.
     *
     * @return the size
     */
    public int getSize() {
        return this.individuals.size();
    }

    /**
     * randomly get an individual.
     *
     * @return the individual
     */
    public I getIndividual() {
        return this.individuals.get(this.random.nextInt(this.individuals.size()));
    }

    /**
     * get the individual with given index.
     *
     * @param index the index
```

```
	 * @return the individual
	 */
	public I getIndividual(int index) {
		return this.individuals.get(index);
	}

	/**
	 * https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html
	 *
	 * @param individualComparator the individual comparator
	 */
	public void sortPopulation(Comparator<I> individualComparator) {
		Collections.sort(this.individuals, individualComparator);
	}

	/**
	 * for trim of a list, use sublist to get and return only
	 * a given length list of the original list.
	 *
	 * @param length the length
	 */
	public void trim(int length) {
		this.individuals = this.individuals.subList(0, length);
	}

}
```

### 1.2.11 ReadData.java

```
package SRGP;

import java.io.*;
import java.util.*;

// TODO: Auto-generated Javadoc
/**
 * The Class ReadData.
 */
public class ReadData {

	/** The dependent. */
	ArrayList<Double> dependent = new ArrayList<Double>();

	/** The independent. */
	ArrayList<Double> independent = new ArrayList<Double>();
```

```
    /** The file name. */
    String fileName;

    /**
     * Instantiates a new read data.
     *
     * @param fileName the file name
     */
    public ReadData(String fileName) {
        this.fileName = fileName;
    }

    /**
     * Read.
     *
     * @return the double[][]
     */
    public Double[][] read() {
    try {
        BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream(this.fileName)));
        String str;
        while (( str = br.readLine()) != null) {
         // for sr_line
         //String[] xAndY = str.split("  ");
         // for circle file, just manually to make the data separated by two spaces
         String[] xAndY = str.split("  ");
         //System.out.println(xAndY.length + " " + i++);
         //TODO
         //using a while loop to read the data without manually manipulating the data
file
         double x = Double.valueOf(xAndY[1]);
         this.independent.add(x);
         double y = Double.valueOf(xAndY[2]);
         this.dependent.add(y);
        }
        br.close();
    } catch (IOException e) {
        System.out.println("ERROR: unable to read file " + fileName);
        e.printStackTrace();
    }

    int dataPoints = this.dependent.size();
    Double[][] input = new Double[dataPoints][2];
    for (int i = 0; i < dataPoints; i++) {
        input[i][0] = this.independent.get(i);
        input[i][1] = this.dependent.get(i);
    }
    return input;
    }

}
```

1.2.12 SRFitness.java

```
package SRGP;

// TODO: Auto-generated Javadoc
/**
 * The Class SRFitness.
 */
class SRFitness implements Fitness <GPIndividual, Double> {

    /** The input. */
    private Double[][] input;

    /** The rows. */
    private int rows;

    /**
     * Instantiates a new SR fitness.
     *
     * @param input the input
     */
    public SRFitness(Double[][] input) {
        this.input = input;
        rows = input.length;
    }

    /** The count. */
    public static long count = 0;

    /* (non-Javadoc)
     * @see SRGP.Fitness#calculate(SRGP.Individual)
     */
    public Double calculate(GPIndividual individual) {
        count++;
        Node n = individual.getTree();
```

```java
		return this.fitness(n);
	}

	/**
	 * to calculate the fitness for one (binary tree).
	 *
	 * @author wangsong
	 * @param n the n
	 * @return the double
	 * @param: input data, one row per data point, follow the
	 * format "  x  y" separated by two spaces
	 * @return: the total absolute error between calculated y and
	 * the value in the input per data point
	 */
	public double fitness(Node n) {
		double diff = 0;
		Vocab v = new Vocab();
		for (int i = 0; i < rows; i++) {
			double x = input[i][0];
			v.setVariable(x);
			double y = input[i][1];
			double calculated = n.evaluate(v);
			diff  = diff + Math.abs(y - calculated);
		}
		return diff;
	}

}
```

### 1.2.13 Vocab.java

```java
package SRGP;

import java.util.*;

// TODO: Auto-generated Javadoc
/**
 * this class stores all the operators or variables that will be used
 * and can be viewed as the material to be used for building the binary tree
 * or lets say the vocabulary for the binary tree.
 *
 * @author wangsong
 */

public class Vocab {
```

```java
/** The x. */
// x holds the value for the variable
private double x = 0;

/** The random. */
// a random object to randomly choose operators
Random random = new Random();

/** The non terminal operators. */
private ArrayList<Operator> nonTerminalOperators = new ArrayList<Operator>();

/** The terminal operators. */
private ArrayList<Operator> terminalOperators = new ArrayList<Operator>();

/** The operators. */
private ArrayList<Operator> operators = new ArrayList<Operator>();

/**
 * Instantiates a new vocab.
 */
public Vocab() {
    operators.add(Operators.ADD);
    operators.add(Operators.SUB);
    operators.add(Operators.MUL);
    operators.add(Operators.DIV);
    operators.add(Operators.VARIABLE);
    operators.add(Operators.CONSTANT);
    for (Operator o : operators) {
        if (o.numArg() == 0) {
            this.terminalOperators.add(o);
        } else {
            this.nonTerminalOperators.add(o);
        }
    }
}

/**
 * Gets the variable value.
 *
 * @return the variable value
 */
public double getVariableValue() {
    return x;
    }

    /**
     * Sets the variable.
     *
     * @param value the new variable
     */
    public void setVariable(double value) {
        x =  value;
    }

    /**
     * Gets the operators.
     *
     * @return the operators
     */
    public List<Operator> getOperators() {
        return this.operators;
    }

    /**
     * Gets the one non terminal operator.
     *
     * @return the one non terminal operator
     */
    public Operator getOneNonTerminalOperator() {
        return
this.nonTerminalOperators.get(this.random.nextInt(this.nonTerminalOperators.size()));
    }

    /**
     * Gets the one terminal operator.
     *
     * @return the one terminal operator
     */
    public Operator getOneTerminalOperator() {
        return
this.terminalOperators.get(this.random.nextInt(this.terminalOperators.size()));
    }
}
```