

Genetic Programming

V. Hunter Adams

I've used a tree structure to implement a genetic program that evolves populations of mathematical functions. Below, I've used the program to fit three sets of data (an asymptotic division and a line) both with and without noise. Fitting a circle became a very hard lesson in the importance of representation, which is explained in Section D.

A. REPRESENTATION

Fundamentally, my GP manipulated objects of two classes that I defined: person and population. Each person is a tree of constants, operators, and variables; and the population is a list of persons. Both the person class and the population class had parameters that I could manipulate from run to run.

The knobs that I was able to turn (on the person level) included operator mutation rate (the rate at which an operator would randomly change into another operator), constant mutation rate (the rate at which constants turned into other constants), variable introduction rate (the rate at which constants turned into variables), variable outroduction rate (rate at which variables turned into constants), maximum constant (maximum allowable value for a constant), maximum depth (maximum depth of tree), and extension rate (rate at which the tree tends to grow new branches). I found some combinations of parameters that worked better than others (see below).

On the population level, I had the ability to vary elitism (the proportion of elite individuals that I kept from each generation), selection pressure, and population size.

However, this representation did lead me to some problems. Whenever the tree found the variable 'x', it would return an array of all the x points from the data files and perform all the operations in the tree on that array of x's. For explicit functions of the form $y = f(x)$ (like the div or the line), this worked very well and the GP found the solution without any problem. The same was not the case for the circle.

I will go into more depth in the Circle section of this report, but I had a huge amount of trouble getting the square root operator to work. I backed myself into a corner by using representation that demanded equations of the form $y = f(x)$, and I had trouble getting myself out of that corner. See Section D for more explanation.

To evaluate fitness, I performed a mean-square-error operation between the y-points that the GP produced and the given y-points.

B. THE ASYMPTOTIC DIVISION

1. Without Noise

For the asymptotic division, the GP found a solution that generated points that were indistinguishable from the target solution after 125,000 evaluations. I ran the GP a few times, turning the knobs mentioned above for each run. See the plots below for the optimal solution that the GP found and the fitness vs. evaluations plots for each run.

For the division without noise, the GP found the solution, with mean square error of order 10^{-3}

$$y = x + \frac{0.0257}{x} \tag{1}$$

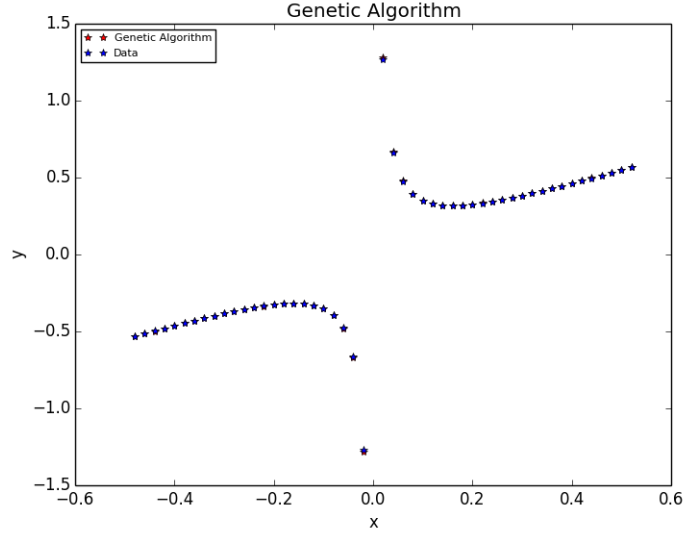


FIG. 1: Plot of the given data points and the genetically produced data points. The GP had an elitism pressure of 0.3, selection pressure of 0.4, operator mutation rate of 0.5, constant mutation rate of 0.5, variable introduction rate of 0.6, variable outroduction rate of 0.4, max depth of 100, and extension rate of 0.7. What all of that means is that the GP preferred to add variables as opposed to take them away, and it preferred to create larger trees as opposed to smaller ones.

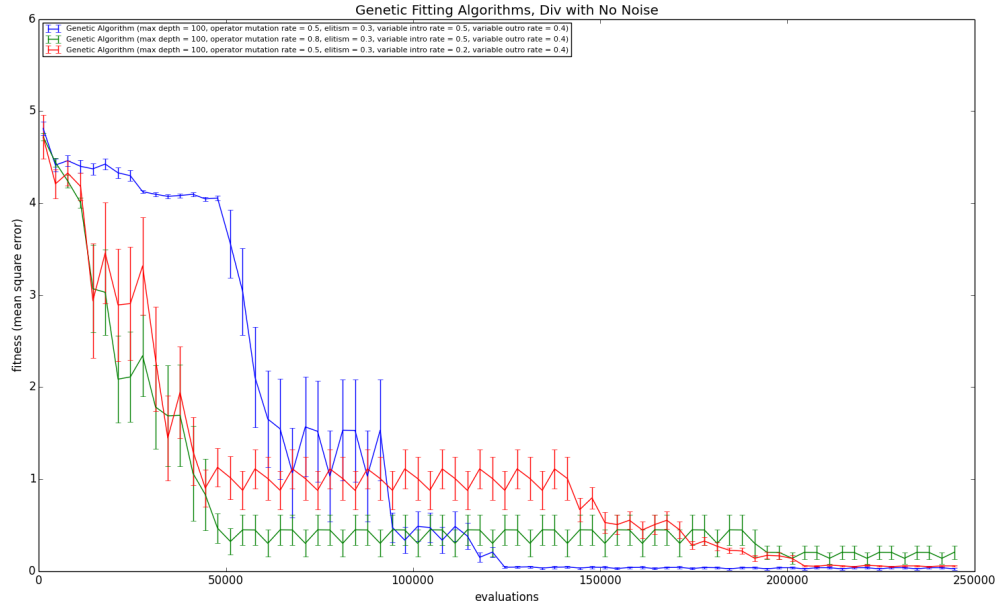


FIG. 2: Fitness vs. number of evaluations for multiple runs of the GP on the asymptotic division with no noise, with different knobs turned different directions. Here, I varied the operator mutation rate and the variable introduction rate. In the long term, I found that the GP with variable introduction rate of 0.5, variable outroduction rate of 0.4, and operator mutation rate of 0.5 won over the GP's that were more reluctant/excited to add variables or mutate operators.

2. With Noise

I noticed some interesting behavior with the noisy asymptotic division (and, in fact, for the noisy functions in general). The GP would place noise into some of the optimal solutions, but the best solution that it returned (the one with the lowest mean square error) was not always a noisy one. This could potentially be because I used a gaussian distribution to add noise to the function, and I set the standard deviation of that gaussian noise to 0.2 (based on inspection). It is possible that the noise that I was adding did not exactly match the type of noise in the function, and that this is why the fittest solutions were not always noisy.

I used the GP to find a solution without any domain knowledge at all, and it returned a very deep tree that fit the data very well. However, when I introduced a little bit of domain knowledge from the solution of the noiseless division (simply by restricting the depth of the tree), the search space was reduced and it eventually found a better solution. Though the **best** solution still did not include noise, which is clearly a problem.

The fittest solution that the GP found for the noisy data was:

$$y = x + \frac{0.0256}{x \cos(x)} \quad (2)$$

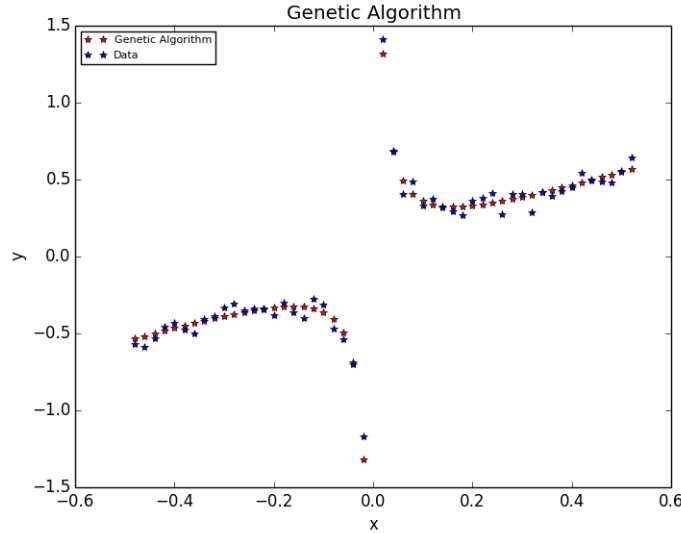


FIG. 3: Plot of the given data points and the genetically produced data points. The GP had an elitism pressure of 0.3, selection pressure of 0.4, operator mutation rate of 0.5, constant mutation rate of 0.5, variable introduction rate of 0.6, variable outroduction rate of 0.4, max depth of 100, and extension rate of 0.7. What all of that means is that the GP preferred to add variables as opposed to take them away, and it preferred to create larger trees as opposed to smaller ones.

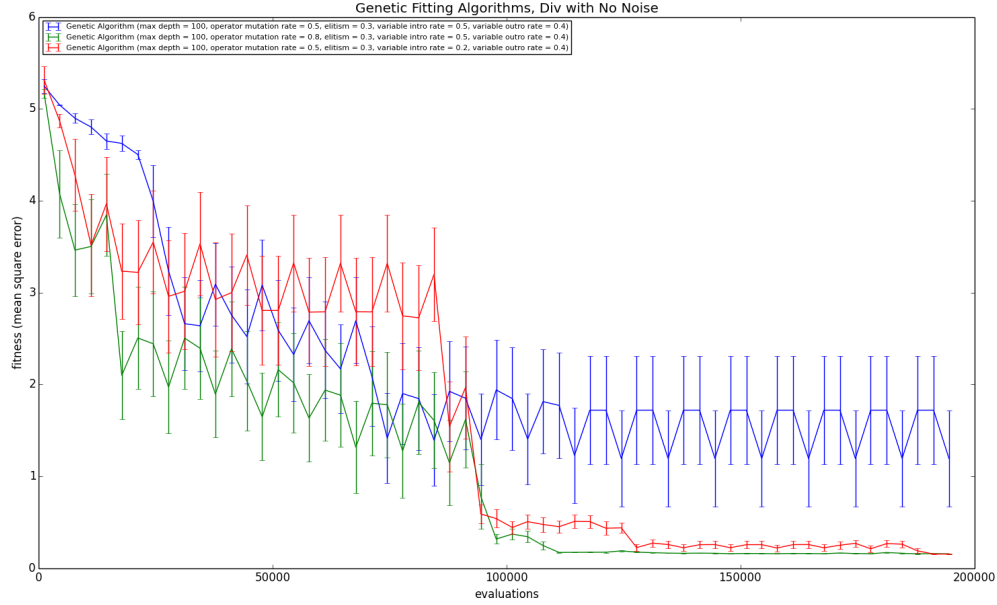


FIG. 4: Fitness vs. number of evaluations for multiple runs of the GP on the asymptotic division with noise, with different knobs turned different directions. Here, I varied the operator mutation rate and the variable introduction rate. In the long term, I found that the GP with variable introduction rate of 0.5, variable outroduction rate of 0.4, and operator mutation rate of 0.8 won over the GP's that were more reluctant/excited to add variables or mutate operators. This differs from the noiseless function, which did not prefer to have its operators mutated as frequently.

C. THE LINE

3. Without Noise

The noiseless line seemed to be too easy of a problem for the GP. In fact, my program would initialize itself by generating a population of 50 equations, each of depth two. I would often find that ranking this initial (random) population often returned a solution that was already pretty close to optimal. Within just a few generations, the GP found a solution that generated y points that were completely indistinguishable from the given y points (mean square error on the order 10^{-4}) after 1500 evaluations. I included fitness vs. evaluations plots for the line, but they are not particularly interesting to look at because this was such a simple problem for the GP.

The GP found the solution:

$$y = e^{\frac{0.904}{2.23} x} = 1.5x \quad (3)$$

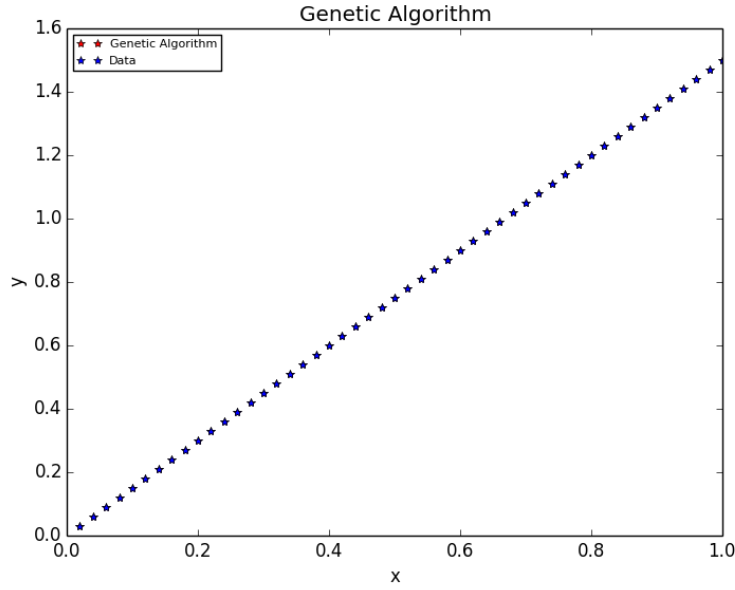


FIG. 5: Plot of the given data points and the genetically produced data points. The GP had an elitism pressure of 0.3, selection pressure of 0.4, operator mutation rate of 0.5, constant mutation rate of 0.5, variable introduction rate of 0.6, variable outroduction rate of 0.4, max depth of 100, and extension rate of 0.7. What all of that means is that the GP preferred to add variables as opposed to take them away, and it preferred to create larger trees as opposed to smaller ones.

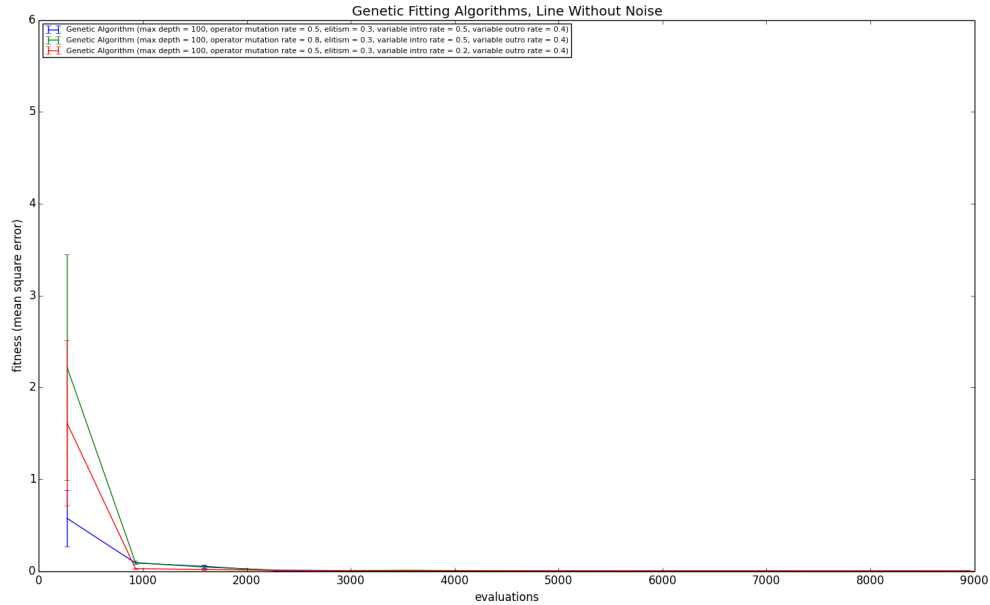


FIG. 6: Fitness vs. number of evaluations for multiple runs of the GP on the line with no noise, with different knobs turned different directions. As I mentioned, the line does not generate particularly interesting data.

4. With Noise

I noticed the same phenomenon with the noisy line that I noticed with the noisy asymptotic division, where the best solution generated did not include the 'noise' operator. As I mentioned previously, this could potentially be because I was adding the wrong type of noise (perhaps it was not gaussian, or perhaps the standard deviation was not 0.02). The GP still found a solution with a mean square error of 0.14, given by:

$$y = x + (0.122 * 8.56) \sin\left(\frac{x}{2.03}\right) = x + 1.04 \sin\left(\frac{x}{2.03}\right) \quad (4)$$

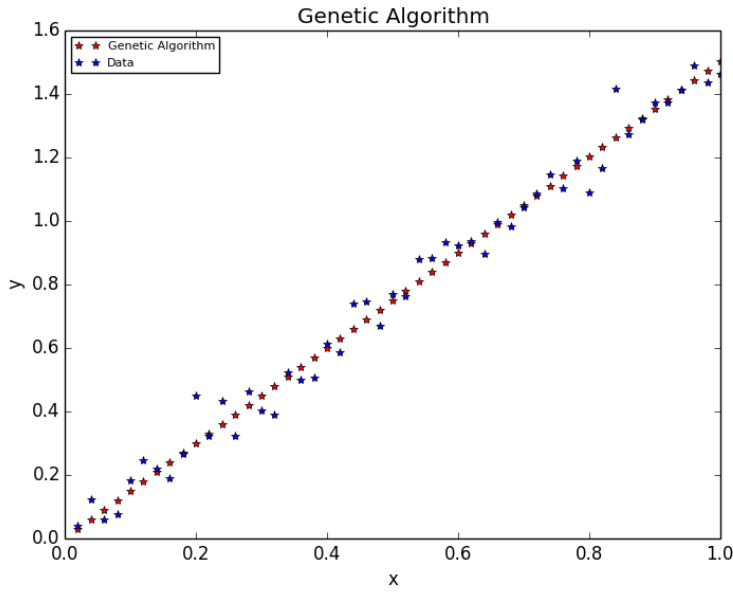


FIG. 7: Plot of the given data points and the genetically produced data points. The GP had an elitism pressure of 0.3, selection pressure of 0.4, operator mutation rate of 0.5, constant mutation rate of 0.5, variable introduction rate of 0.6, variable outroduction rate of 0.4, max depth of 100, and extension rate of 0.7. What all of that means is that the GP preferred to add variables as opposed to take them away, and it preferred to create larger trees as opposed to smaller ones.

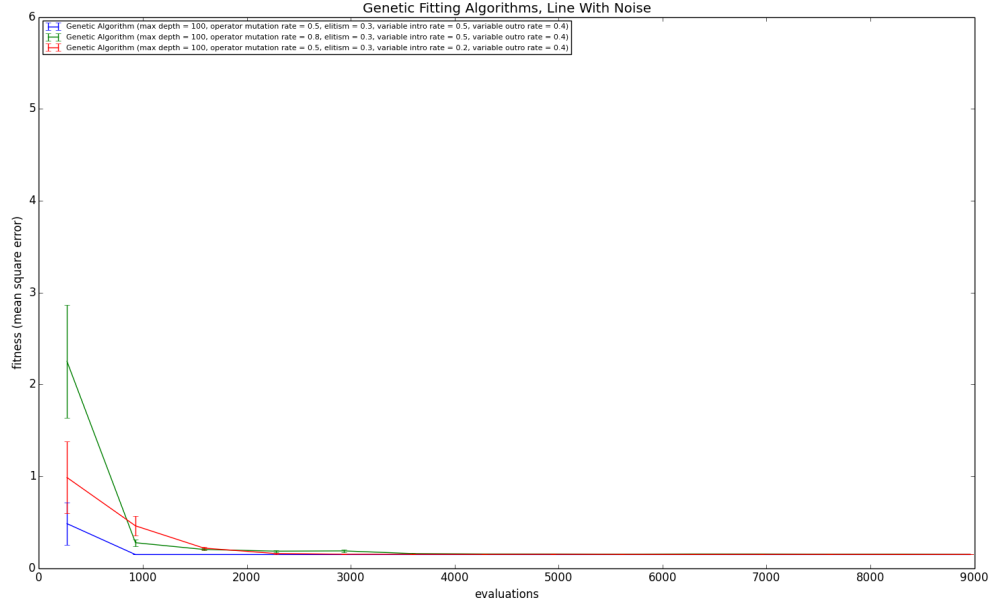


FIG. 8: Fitness vs. number of evaluations for multiple runs of the GP on the line with noise, with different knobs turned different directions. As I mentioned, the line does not generate particularly interesting data.

D. THE CIRCLE: PROBLEMS ENCOUNTERED AND ATTEMPTED SOLUTIONS

As I mentioned at the beginning of this report, I backed myself into a corner by choosing a representation of the form $y = f(x)$ for my GP. Below, I've explained each problem that I encountered and the solution that I implemented to solve that problem. In the end, the GP did not find an optimal solution for the circle. I don't feel like I wasted my time, however, because wrestling with this problem taught me a lot of lessons the (very) hard way.

5. Problem: Square Root

Because I chose the representation that I did, the equation for the circle was not

$$x^2 + y^2 = 1 \quad (5)$$

but was instead:

$$y = \pm \sqrt{1 - x^2} \quad (6)$$

In theory, this is fine as long as the square root operator knows how to use \pm . Naively implementing square root resulted in just the positive square root, so I needed a way to get both positive and negative roots.

6. Attempted Solution: Linkage

The solution for this problem that I came up with was to sort the data according to x value before passing it to the GP, and then to multiply the array that I get from the square root operator (which contains the positive square roots of each x point passed to it) by an array of the form $[-11 - 1 \dots]$.

The square root function is not 1 to 1 (it returns two values for every value that it is passed). **However**, any data set contains lists of x and y coordinates where every x has a corresponding y . Some of those x 's may be of the same value (as is the case for the square root), but every point is specified by a single x and a single y .

I attempted to exploit this fact when building the square root operator (this also used some domain knowledge about the data that I was fitting, i.e. that it was a circle). By arranging the data according to x value, I could perform the square root operation on the entire array and then negate every other component of the array. When I did this, I got plots like the one shown below which **look** like proper plots of square root.

There are a couple problems with this solution. The first is that I exploited some domain knowledge in constructing my square root operator, and the second is that the GP refused to evolve the functions beyond what is shown below. For the circle, I seem to have become a victim of my representation. A more elegant solution would have been to represent the equations in the form $f(x, y) = 0$ and then to evaluate fitness symbolically as opposed to numerically.

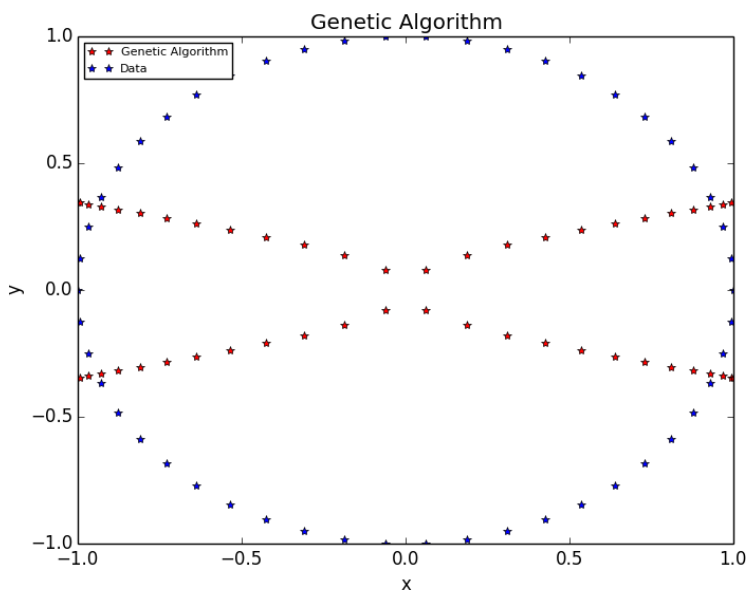


FIG. 9: The best that my GP could do with the circle. By appearance, the square root operator appears to be working. Because I chose the wrong implementation for this problem, the GP has tremendous trouble finding the solution.

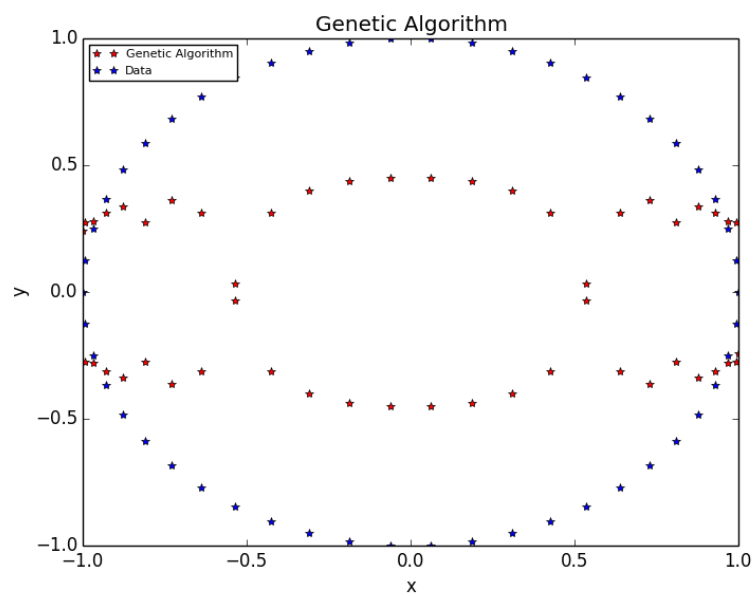


FIG. 10: This is another attempt that appeared promising, then stagnated.

E. MY CODE

```
from numpy import *
from scipy import *
from operator import itemgetter
from random import sample
from ast import *
import matplotlib.pyplot as plt
from copy import deepcopy

operator_dict = {-1: 'sqrt', 0: 'noise', 1: '+', 2: '-', 3: '*', 4: '/', 5: 'sin', 6:
    'cos', 7: 'exp', 8: 'x'}

evaluations = 0
flipper=[]
for i in range(50):
    if i%2==1:
        flipper.extend([-1])
    else:
        flipper.extend([1])
flipper=array(flipper)

class person(object):
    def __init__(self, root = None, left = None, right = None):
        self.root = root
        self.left = left
        self.right = right

    def evaluate(self,dic):
        try:
            if self.root == operator_dict[-1]:
                return
                self.left.evaluate(dic)*(abs(sqrt(self.right.evaluate(dic)))*flipper)

            elif self.root == operator_dict[0]:
                r = []
                for i in range(len(array(self.right.evaluate(dic)))):
                    y = list(random.normal(1,0.05,1))
                    r.extend(y)
                return self.left.evaluate(dic) * (r*array(self.right.evaluate(dic)))
            elif self.root == operator_dict[1]:
                return self.left.evaluate(dic) + self.right.evaluate(dic)
            elif self.root == operator_dict[2]:
                return self.left.evaluate(dic) - self.right.evaluate(dic)
            elif self.root == operator_dict[3]:
                return self.left.evaluate(dic) * self.right.evaluate(dic)
            elif self.root == operator_dict[4]:
                return self.left.evaluate(dic) / self.right.evaluate(dic)
            elif self.root == operator_dict[5]:
                return self.left.evaluate(dic) * sin(self.right.evaluate(dic))
            elif self.root == operator_dict[6]:
                return self.left.evaluate(dic) * cos(self.right.evaluate(dic))
            elif self.root == operator_dict[7]:
                return self.left.evaluate(dic) * exp(self.right.evaluate(dic))
```

```

        elif self.root == operator_dict[8]:
            return array(map(lambda dic: dic[0], dic))
        else:
            return self.root

    except:
        return nan

def depth(self):
    left_depth = self.left.depth() if self.left else 0
    right_depth = self.right.depth() if self.right else 0
    return max(left_depth, right_depth) + 1

def find_branch(self, depth):
    level = 0
    if level==depth:
        dice = random.randint(1,3)
        if dice == 1:
            return self.right
        else:
            return self.left
    else:
        level+=1
        dice = random.randint(1,3)
        if dice == 1:
            try:
                return self.left.find_branch(depth)
            except:
                pass
            try:
                return self.right.find_branch(depth)
            except:
                pass
            return person(self.root)
        if dice == 2:
            try:
                return self.right.find_branch(depth)
            except:
                pass
            try:
                return self.left.find_branch(depth)
            except:
                pass
            return person(self.root)

def mother_chromosome(self, depth):
    chromosome = ['child']
    level = 0
    if level==depth:
        dice = random.randint(1,3)
        if dice == 1:
            chromosome.extend(['.right'])
            return ''.join(chromosome)
        else:
            chromosome.extend(['.left'])

```

```

        return ''.join(chromosome)
    else:
        level+=1
        dice = random.randint(1,3)
        if dice == 1:
            try:
                chromosome.extend(['.left'])
                return self.left.mother_chromosome(depth)
            except:
                pass
            try:
                chromosome.extend(['.right'])
                return self.right.mother_chromosome(depth)
            except:
                pass
            return ''.join(chromosome)
        if dice == 2:
            try:
                chromosome.extend(['.right'])
                return self.right.mother_chromosome(depth)
            except:
                pass
            try:
                chromosome.extend(['.left'])
                return self.left.mother_chromosome(depth)
            except:
                pass
            return ''.join(chromosome)

def fitness(self,dic):
    evals = evaluations
    evals = evals + 1
    global evaluations
    evaluations = evals
    try:
        genotype = self.evaluate(dic)
        if type(genotype)==float64:
            return 10000000.
        elif type(genotype)==ndarray and math.isnan(genotype[0]):
            return 10000000.
        elif type(genotype)==ndarray and len(genotype)<5:
            return 10000000.
        elif type(genotype)==float:
            return 10000000.
        else:
            value = sum((self.evaluate(dic)-\
                array(map(lambda dic: dic[1],dic)))*2)
            return value
    except:
        print "Couldn't evaluate fitness"
        return 1000000000.

def crossover(self,spouse):
    max_depth = min(self.depth(),spouse.depth())
    depth = random.randint(1,max_depth)

```

```

        child = deepcopy(self)
        father = '=spouse.find_branch(depth)'
        mother = child.mother_chromosome(depth)
        DNA = mother+father
        exec(DNA)
        return child

def
    mutate(self,operator_mutation_rate,constant_mutation_rate,variable_intro_rate,variable_outro_rate)
    if type(self.root)==str and self.root!=operator_dict[8]:
        dice = random.random()
        if dice < operator_mutation_rate:
            op = random.randint(-1,8)
            self.root = operator_dict[op]
            self.left.mutate(operator_mutation_rate,\
                constant_mutation_rate,variable_intro_rate,variable_outro_rate,\
                extension_rate,max_constant,maximum_depth)
            self.right.mutate(operator_mutation_rate,\
                constant_mutation_rate,variable_intro_rate,variable_outro_rate,\
                extension_rate,max_constant,maximum_depth)
        else:
            self.left.mutate(operator_mutation_rate,\
                constant_mutation_rate,variable_intro_rate,variable_outro_rate,\
                extension_rate,max_constant,maximum_depth)
            self.right.mutate(operator_mutation_rate,\
                constant_mutation_rate,variable_intro_rate,variable_outro_rate,\
                extension_rate,max_constant,maximum_depth)
    elif type(self.root)==float:
        dice = random.random()
        dice1 = random.random()
        dice2 = random.random()
        if dice < constant_mutation_rate:
            const = random.uniform(self.root-0.5*self.root,self.root+0.5*self.root)
            self.root=const
        if dice1 < variable_intro_rate:
            self.root=operator_dict[8]
        if dice2 < extension_rate and self.depth()<maximum_depth:
            op = random.randint(-1,8)
            self.root = operator_dict[op]
            self.right = person(random.uniform(0,max_constant))
            self.left = person(random.uniform(0,max_constant))
    elif self.root==operator_dict[8]:
        dice = random.random()
        if dice < variable_outro_rate:
            self.root = random.uniform(0,max_constant)
        else:
            pass

class population(object):
    def __init__(self,people = None):
        self.people=[people]

    def add_person(self,person):
        self.people.extend([person])

```

```

def rank(self,dic):
    caliber = []
    for i in self.people:
        caliber.extend([i.fitness(dic)])
    rating = sorted(zip(caliber,self.people),key=itemgetter(0))
    self.people = map(lambda rating: rating[1], rating)

def select(self,elite_pressure,total_pressure):
    parent_population=[]
    elite = int(elite_pressure*len(self.people))
    parent_population.extend(self.people[0:elite])
    remaining = int(total_pressure*len(self.people))-elite
    for i in range(elite,elite+remaining):
        parent = random.randint(elite,len(self.people))
        parent_population.extend([self.people[parent]])
    self.people = parent_population

def
    breed(self,population_size,operator_mutation_rate,constant_mutation_rate,variable_intro_rate,var
    parents = len(self.people)
    num_children = population_size - parents
    for i in range(num_children):
        mother = self.people[random.randint(0,parents)]
        father = self.people[random.randint(0,parents)]
        child = mother.crossover(father)
        child.mutate(operator_mutation_rate,constant_mutation_rate,\
            variable_intro_rate,variable_outro_rate,extension_rate,max_constant,\
            maximum_depth)
        self.add_person(child)

def plotter(self,dic):
    x1=[]
    y1=[]
    x2=[]
    y2=[]
    for i in range(len(dic)):
        x1.extend([dic[i][0]])
        x2.extend([dic[i][0]])
        y1.extend([dic[i][1]])
    y2=list(self.people[0].evaluate(dic))
    plt.plot(x2,y2,'r*',label="Genetic Algorithm")
    plt.plot(x1,y1,'b*',label="Data")
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Genetic Algorithm')
    plt.legend(loc='upper left',prop={'size':8})
    plt.show()

def
    evolve(self,dic,generations,elite_pressure,total_pressure,population_size,operator_mutation_rate
    for i in range(generations):
        print 'Generation Number: '+str(i)
        self.rank(dic)
        print self.people[0].fitness(dic)

```

```
print evaluations
print '\n'
self.select(elite_pressure,total_pressure)
self.breed(population_size,operator_mutation_rate,\
    constant_mutation_rate,variable_intro_rate,variable_outro_rate,\
    extension_rate,max_constant,maximum_depth)
```
