Approximating ODE Solutions Using Genetic Programming

Martin Schmidt

May 14, 2010

Abstract

Many ordinary differential equations (ODE) are without closed-form solutions. While numerical methods such as Runge-Kutta and Euler can provide accurate point instance mapping using a model of the solution, they are limited without an explicit form. One way around this is to optimize a closed form approximation. Genetic Programming (GP) is an optimization method we can use to find an explicit approximation for many ODEs with or without a closed form solution. We will look at how to implement a genetic program to approximate solutions to ODEs with and without closed-form solutions.

1 Introduction

We witness the process of creative destruction in everyday life. For instance, business models that succeed tend to survive and are adopted by others while those that fail fall by the wayside. We know that propping up bad businesses is poor economic strategy. Creative destruction also is necessary for the process of natural selection to work. Our entire existence depends not only on our ancestors ability to survive, but also on the death of others and the extinction of species. (We would not be here if the dinosaurs among others did not go extinct). Whether this process occurs naturally or artificially the logic is the same. Genetic programming uses this idea of selection of the best fit to find optimal solutions to problems. We will examine how to use genetic programming to solve an initial value problem of the first-order and also how to approximate an ODE with no closed-form solution.

2 Solutions to First-Order ODEs

A general first-order ordinary differential equation (ODE) can be described by the equation $y' = f(x, y)$, where y' is a function of an independent variable, x, and a dependent variable, y. Our interest is in finding a unique differentiable function $y = f(x)$ that satisfies the differential equation given an initial value

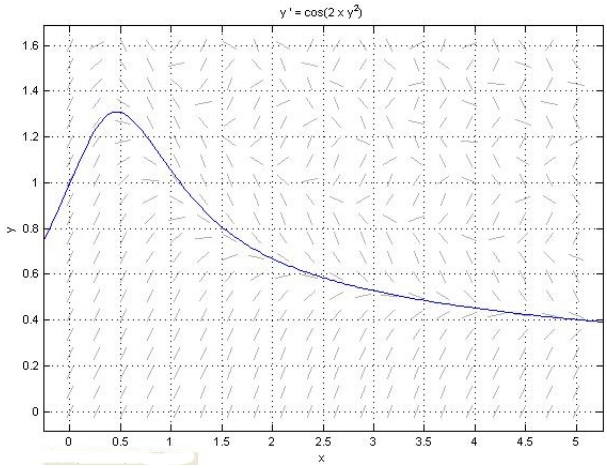


Figure 1: Example of dfield8, a direction field and a numerical solution to a ODE

over some domain. Correct modeling of a system requires the uniqueness of our solution, since two solutions would lead to contradictions. So as we set up our equations, the domain we use must be viable. If our function $y' = f(x, y)$ and its partial derivative over y are both continuous over our region of interest, the solution $y = f(x)$ will be unique.

There are analytical methods to solve certain kinds of first-order ODEs. For example, if our equation can be rewritten so that $f_1(y)dy = f_2(x)dx$, we may integrate both sides with respect to their variable solve for y (Separable Method). If the function $y' = f(x, y)$ contain only linear terms of y such that we can rewrite our equation as $y' + f_1(x)y = f_2(x)$, the solution is $y = e^{-\int f_1(x)dx} \times \int f_2(x)dx$ (Linear Method). If our ODE can be rewritten in one of these forms, we can be guaranteed a closed-form analytic solution. But often a closed-form solution will not exist for our first-order ODE $f' = f(x, y)$.

3 Approximations to First-Order ODEs

When a closed-form solution does not exist, or if the equation is too complicated to put into a form to apply an anlytical method, an approximation of the solution can be performed. Since we can calculate the slope of any point in the xy-plane in our domain from $y' = f(x, y)$, we may set up a direction field to see how our solution should look. We will use dfield8 in Matlab to see how this would work.

If we would like to know what the y approximation may be given certain x values, we may also collect these numerically derived points in an array $[x, y]$ using a numerical solver in Matlab. We will use the numeric solver ode45 in our example.

3.1 Interpolation and Curve-Fitting

The function ode45 takes our differential equation, the domain, and initial value as its parameters and returns the array $[x, y]$, where x has a variable-step and number of values, and y is the corresponding approximation of our solution at these points. We will assume that the error between the y values and the actual values of the solution are negligible (though this will depend on the rigidity of the system).

We can use the points in $[x, y]$ to derive an analytic approximation that withstands a certain tolerance. If the system under examination is not very complicated (ie doesn't contain many turns), the length of [x,y] vector may be relatively short. Then we may want to approximate the solution by interpolation of a polynomial since given n points, there is an unique n-1 degree polynomial that fits each of these points exactly. Matlab provides the functions, polyval and polyfit that will give us the coefficients of the n degree polynomial.

One advantage to polynomial interpolation/regression is that they provide equations that are easy to differentiate/integrate. We all know how to work with polynomials. But if the number of data points is substantial and the best polynomial fit is of an equally substantial nth power, then we may abandon using polynomial interpolation/regression. We may want to try other types of regression, but using human judgement as to what type of regression would fit best is often not the best approach.

4 Genetic Programming

Could we automate a search for the best fit of our solution? Let's say we have a bag of building blocks that the program could randomly take out and use to build a function. Obviously, it would need a bag of real number blocks and independent variable blocks. But simply using constant numbers and an independent variable won't get us very far. We need function blocks as well. So if we include a bag of binary functions, such as addition $(+)$, subtraction $(-)$, and multiplication $(\times)$ along with our real number blocks, it could build any polynomial (we assume our bag is infinite in size). But that's what we were doing with polynomial interpolation/regression. The whole point of our program is to find a different kind of function. So we'll also include division $(\div)$. Now we can construct inverse functions along with our polynomial (but all the while being careful not to divide by zero).

Now if our program were just to randomly pull out numbers and functions, scrap the solution if it didn't meet a tolerance, start from scratch and randomly pull out numbers and functions, scrap the solution...and so on, it is very unlikely we would ever randomly make a function that fits the solution points very well. It would be like having the parts to a watch laying about in a garage and a tornado coming in to assemble it into an airplane (or however that argument goes). However, natural selection provides a way around this.

Genetic programming (GP) uses the logic of natural selection to produce

algorithms that optimize approximate solutions to problems. GP is best used when the properties of the solution are known but the solution itself is not. For example, if you wanted to maximize the aerodynamicity of a surface, you could apply GP. You know that the solution needs to produce the lease amount of drag, but you don't know what surface does that. The genetic information you would use could be an equation representing the surface shape in a container. You'd create a population of equations, numerically test each one in a virtual wind chamber, evaluate the equations, select the best fit and pass the information to a next generation via reproduction (crossover or mutation) and repeat the process until you got a surface that fits your requirements.

However, GP should not be used for problems where there exists an analytical method for solving the problem. If there exists an exact solution and you know how to get it, why would you need to optimize an approximate solution? We will give an example of finding a solution to a differential equation with a closed-form solution and compare its results to the method dsolve in Matlab.

4.1 Interpreter and Syntactically-consistent Data

When a person interprets a language, they need to differentiate between words. Apersonmaynotbeabletoreadthiswithoutsometrouble, but you will have no trouble reading the latter part of this sentence. Likewise, a computer must know what kind of words it is interpreting. If a number is represented by 8 bits, we want the computer to read 8 bits as a number. If we have represented an 8 bit number, but the computer reads only 4 bits, there is miscommunication.

This process of reading in our program we will call interpretation and the part of the program that does this will be called the interpreter. To avoid inconsistent data, a type of miscommunication, we will use a type of genetic programming that is syntactically-consistent. This means we will conserve the words in the program in their entirety and will not change individual bits as a process of mutation or crossover. So while an integer may be represented by 4 bytes, our program will never separate this into individual bits to manipulate. Instead, it will perform operations on the 4 bytes as if it were a single unit. (However, this does not suggest that genetic programming done on the bit level does not occur).

Our interpreter is implemented in the *run* method of our Tiny GP (see chapter 8 code). The words of our program will be randomly selected integers between 0 and 115. The program interprets each of these values differently. The value 0 is interpreted as x, the values 1 through 4 are preselected values containing π, 5 through 109 are decimal values randomly initialized between -5 and 5, and 110 through 115 are interpreted as our functions (sine, cosine, exponential, add, subtract, multiply, divide). How our interpreter is constructed may affect the performance of our program. For example, we may find that having over one hundred different numerical values compared to only one variable and seven functions isn't a very good mix. In this case, we could change this parameter.

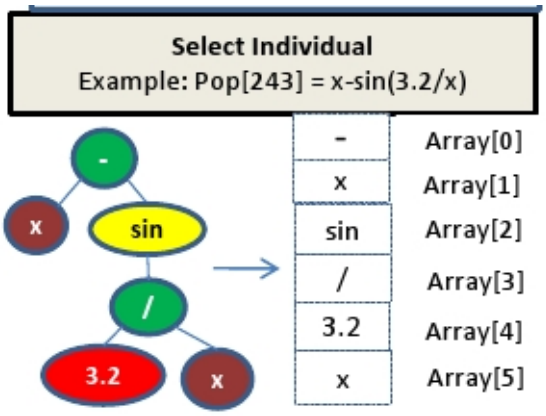


Figure 2: Representing a function as a tree and flattening the tree to an array

4.2 Polish Notation, Trees and Nodes

In everyday use, we represent functions as a string of characters. The operators that act on its arguments, variables and numbers, are infixed. For example, we can apply the operator $\times$ to two arguments, x and 4.23, in the syntactical form $x \times 4.23$. For us, this seems perfectly natural. But we could have just as easily represented the function in a different syntax. For example, we could place the function last and put the two arguments first, so that $x \times 4.23$ becomes $x4.23\times$. Or we can put it first, $\times x4.23$. Since unary operators, such as sin are placed before the argument, always placing the function before its arguments somewhat simplifies its representation. This notation is known as Polish notation. Take for instance the function $f(x) = 3.2 \times (x + \sin(2 - (x \div 4.1))$. We are required to place parenthesis to delimit the orders of operation. If we were to write this in Polish notation, the function becomes $f(x) = \times 3.2 + x \sin -2 \div x4.1$, without the need for parenthesis. This is useful for our purposes because we can now represent our function as a one-dimensional array whose elements are in the same sequence as that of our function in Polish notation.

We can visualize our functions by representing them as trees. Each trees contain nodes and lines connected nodes to each other. The rules for constructing our function tree our simple. If the node is a binary function, two nodes are connected beneath it. If the node is a unary function, one node is connected. If the node is a variable or number, no further nodes are attached. The tree is complete when all terminal nodes are variables or numbers. Each descendant of a node will lie on a depth of one more than the parent. We will limit our initial individuals to be of depth 5 to minimize the length of our approximate solution.

Unfortunately, we cannot just add our tree into the program. We must first flatten it into a structure the computer program can read. This involves placing each node into an address in a one-dimensional array. The program can

then interpret a branch from the start of the array until it reaches two terminal nodes after a binary function or one terminal node after a unary function. This is done by recursively calling the method (see *grow* method in Java code) until it reaches its terminal condition.

4.3 Creating a Population

Each function tree characterizes an individual within our population. The population size depends on the nature of our problem. But in all genetic programming problems there exists a lower limit to the size of the population where our solution has too great a probability of being suboptimal in a given time period. And if our population is inordinately small we run the risk of decreased variability as individuals with relatively higher fitness quickly dominate the gene pool.

The price of increasing our population is decrease our program's performance in speed. The marginal cost of increasing our population by one individual is decreasing our program's performance linearly (or $Big(O) = n$), while our payoff is not entirely clear. We will attempt to make a rough approximation of the maximum payoff per decreased performance by repeated runs.

In our examples, we will use a population of $1,000$ for ODEs with a closed-form solution (Fast-food ODEs)and $10,000$ for those without a closed-form solution (Homecooked ODEs). This is because the program will be able to quickly converge to a solution when one exists and so doesn't need time to grow and diversify, whereas those without require time to stew. And as with fast-food, genetic programming as a method for solving closed-form problems may seem appetizing to try, but shouldn't be used to excess because other heartier methods exist to find the exact solution.

4.4 Fitness Function and Selection

There are two ways we can determine the fitness of our individual functions in the population in our model. One is to take the derivative of the function and then evaluate our differential equation at each sample points in our domain and sum the errors. The downside to this method is that it is computationally more expensive than simple regression seeing that we must differentiate each of individuals before we calculate the error. If our solution is complex, this could halt our program (consider differentiating nested sine and cosine functions). If our differential equation is not too rigid, we can assume that a numerical solver can provide us with reliable sample points to our solution in our domain. This eliminates the differentiating step.

Symbolic regression will be performed on sample points within our domain. Our sample points will be returned by the method ode45 within Matlab. This usually will give us a non-fixed step size of our independent variable. We may assign a fixed step-size explicitly, but this may lead to errors under certain areas of our solution where its slope is highly variable.

In our program, we store the value of our independent variable, x, as the first entry in the array containing our constant terminal values (randomly selected decimal point numbers between 5 and -5). We evaluate the function (see method *fitnessFunction*) at each sample point by loading the x value into the array and calling the *run* method to evaluate the y value. We then add the absolute value of all the errors and reverse the sign. This allows us to use our tolerance value as an upper bound in order to bail out of the program and give us our solution when this is reached.

4.5 Reproductive Operators

Natural selection wouldn't be possible without variability being introduced into the population. We will use the concepts of genetic crossover and mutation to introduce new variability into our population. We will perform ***crossover*** over entire branches of our function tree by selecting a node and consequently the entire branch under it. It is similar to cutting a branch of a natural tree. Then another function tree's branch is connected to where the cut in the first tree took place to produce a child. ***Mutation*** simply selects a node and replaces it with a node of the same type. For example, if a multiplication node is selected, another binary function is randomly selected to take its place.

The fitness of each individual is used to select which individuals will reproduce. In our *tournament* method we select two random individuals from our population. The one with the higher fitness function becomes our first parent function. We then randomly select either to mutate this parent or perform crossover with a second parent. If crossover is selected, we find the best fit of two randomly selected individuals to be the second parent. We then select random nodes in each parent and make the crossover. Two children will result, but we will only keep the first and throw out the second. If mutation is selected, a node is selected and mutated to a node of the same type. These reproductive operators will be used until our population is filled with a new generation of children and the process starts again.

4.6 Parameters

We must decide certain characteristics of our program such as what our population size should be, or how often should we perform crossover as opposed to mutation, or how many individuals should we include in our tournament. These are our parameters of the program. A change in parameters could drastically improve the performance of our program. How should we decide what these values should be?

The first thing to consider is what our solution may look like. If we know there exists a closed-form solution, we may be comfortable with a smaller ***population size*** since an error of zero exists over the whole domain and it is more likely that the population will contain individuals that will converge to this solution quickly. By decreasing the population size we increase the speed of our program. We will use a population size of $1,000$ for problems with closed-form

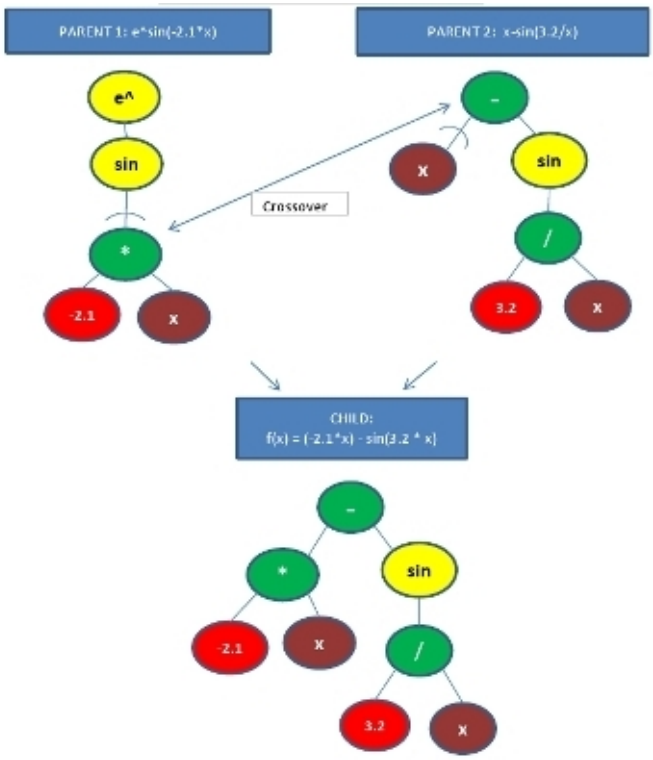


Figure 3: Mechanism of crossover

solutions. For problems with no known closed-form solutions, it is unlikely that any individual will be close to satisfying our tolerance within a small population. So we will use a population size of $10,000$ for problems with no closed-form solutions.

We choose a ***crossover probability*** between 50 to 90 percent and a ***mutation probability*** of one minus this crossover probability. If there are some local optimized peaks in our domain, such that the absolute optimized peak cannot be reached by crossover, we may want to increase our mutation rate which can escape local valleys and find an absolute peak. However, we often will not know that this is the case. Some experimenting with different rates should give the programmer an idea as to what rate to use. It is important to note that if you increase the crossover rate you decrease the speed at which your program will run due to the increased number of nodes to move around and also the bias for increasing the average length of your population. Mutation has the advantage (or disadvantage, depending on the problem) of maintaining the length of the parent.

If you want to limit the length of your solution, the ***maximum length*** of a program is hard-coded and can be decreased. We also use a ***penalty coefficient***

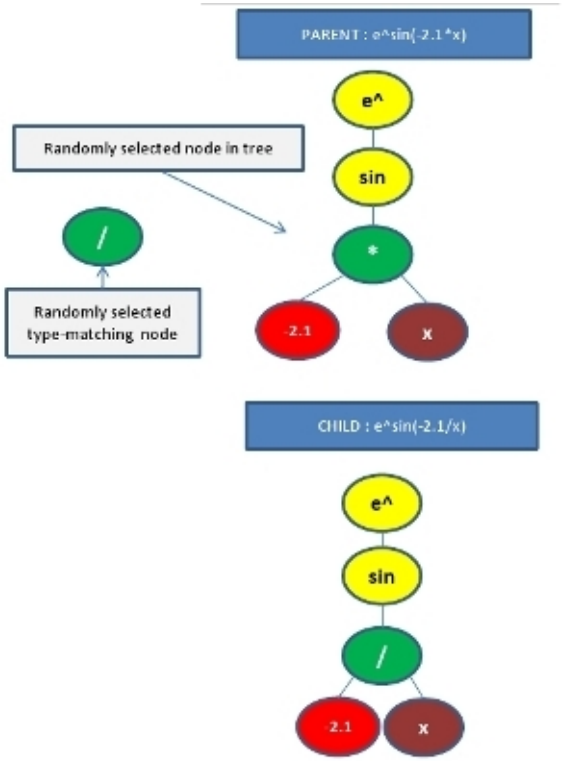


Figure 4: Mechanism of mutation

for solutions when they reach a certain length. Lengthy solutions often occur when a closed-form solution does not exist or the solution is complicated and the tolerance is set too low.

A differential equation with a closed-form solution can handle a ***tolerance***, or error summed over domain, near zero. We will set our tolerance to 0.1 for differential equations with or without closed-form solutions. However, if we have an exorbitant amount of sample points, we may want to increase this number.

If our program cannot find a solution within the defined tolerance, it needs a way to exit gracefully. We do this by setting a ***maximum number of generations***. We will set this number to 600.

There are also a number of ways we could have selected individuals to produce offspring. We chose to perform a ***tournament*** involving two individuals. This runs the risk of selecting two individuals with low fitness. If we were to increase the number of individuals in the tournament we would undoubtedly choose individuals with better fitness for parents, but at the same time lose individuals who increase the diversity of the population. If we lose diversity too quickly, the population becomes stagnant and uniform.

It is possible that different differential equations could have different optimiz-

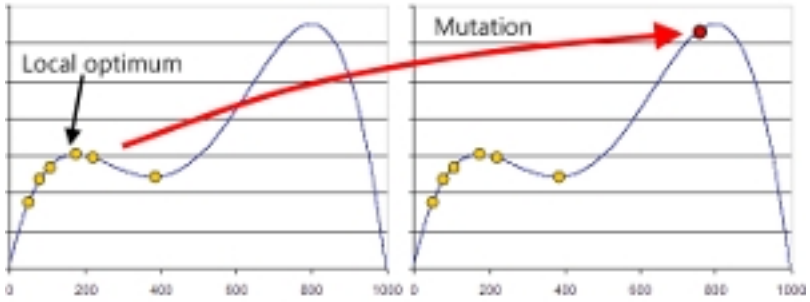


Figure 5: Mutation avoids the pitfalls of crossover

ing parameters within the program. It would be interesting to find the optimal parameter values for each problem (perhaps with a meta-genetic program).

5 Example 1: ODE with Closed-Form Solution

Take the initial value problem $y' = -y \times cos(x)$ at $y(0) = 1$. This is a separable differential equation with its solution being $y = e^{-\sin(x)}$. If we were ignorant of this and wanted to use Matlab's dsolve method, it returns this as an answer and in just under a tenth of a second. There really is no good reason to use genetic programming as an algorithm to solve initial value problems with closed-form solutions, but in the interest of the inner workings of our genetic program (endearingly called Tiny GP, adapted from code originally written by Riccardo Poli), we will try and solve this problem.

Our first step is to set a domain in x under which we will obtain sample points. We will use the domain $[0, 5]$. We then call Matlab's ode45 and send the sample points to a file that will be read by our Tiny GP program which is written in Java.

Next, we set our parameters. Since this problem is known to have a closed-form solution, we will set our population size to $1,000$, which will increase the speed of each generation but decrease variability in our population. We use a crossover rate and mutation rate of 50 percent and keep the rest of the parameters at their default values (see code).

5.1 Example 1: Results

The solution $y = e^{-\sin(x)}$ is found in 31 generations. Although a pseudo-solution, $y = (x \div x) \div e^{\sin(x)}$, is found at generation 20, the protected division by zero returns a value of 0 instead of undefined at $x = 0$. It takes another 11 generations to come across the solution $y = (4.066 \div 4.066) \div e^{\sin(x)}$. Notice that there is no simplification in our genetic program. To maintain valid trees that contain specific terminal values requires it for each generation. However, it may

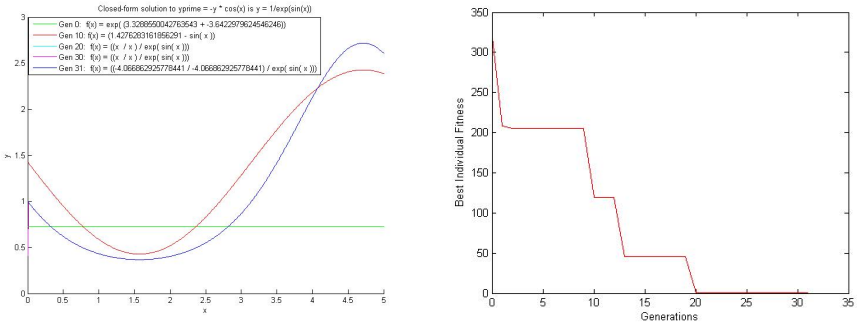


Figure 6: Example 1 Results

be possible to write a routine that could simplify the final solution considering many applications offer simplification functionality (including Matlab).

Although this program didn't outperform dsolve, it didn't do too bad either. It clocked at just over three seconds.

6 Example 2: ODE without Closed-Form Solution

The original goal of this paper was to show how to get an approximate solution to a differential equation with no closed-form solution. If normal regression cannot meet a certain tolerance under a domain, we may want to try symbolic regression using genetic programming. Consider the initial value problem $y' = \cos(2xy)$ at $y(0) = 1$. Ode45 returns 501 sample points in our domain. Running Matlab's polyfit fails and returns an error. So this is a good candidate for using our Tiny GP.

6.1 Example 2: Results

After 375 generations and 1,029 minutes 13 seconds later, we arrive at a good approximation (error of about 0.652) to the differential equation in the domain. The program had the largest error at around $x = 0$. So if we start our approximation at $x = 0.08$, we decrease the error by more than half (error including penalty for length of our function of about 0.2528), which still isn't at our tolerance of 0.1, but considering we calculating the error at each of 501 points, we find the average error at each point being 0.00005 with the maximum error of 0.014.

However good our approximation is, it isn't pretty.

```
 y = e^( ((((x  / cos( sin( ((cos( -1.3977331148909333) / (x  *
((x  - (sin( e^( 0.5839073910575703)) / ((0.5839073910575703 + ((
(cos( -1.0768808474282068) / (((x  - e^( (sin( (x  +
```

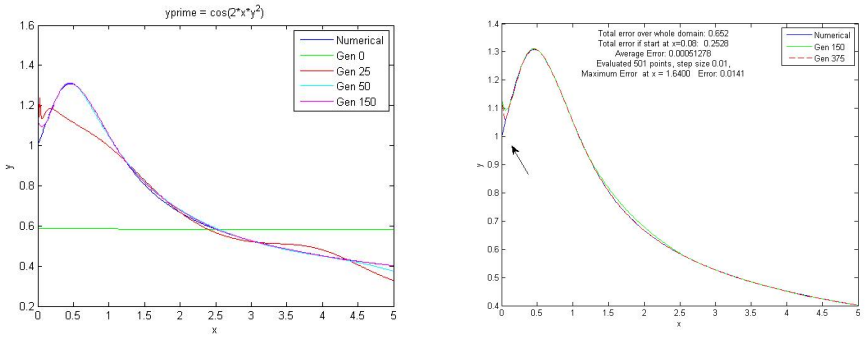


Figure 7: Example 2 Results

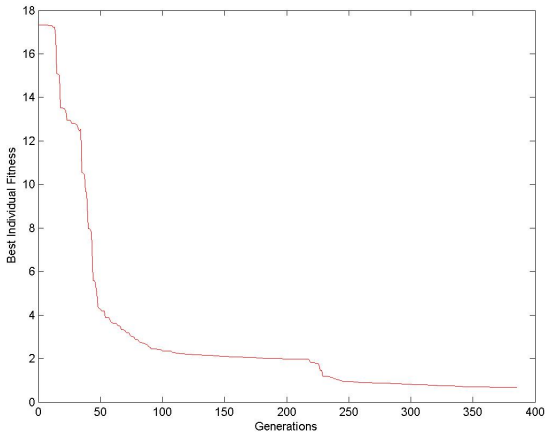


Figure 8: Example 2 Best Fit Individual

```
(-2.1183334622298835 + cos( -1.3977331148909333)))) * (sin( (x
+ 4.404029716618364)) / (x  + x ))))) / -3.9294994016950726) * cos(
cos( cos( ((x  + -2.1183334622298835) / (((-3.488290785681106 /
4.429086745576107) - (4.64613256685335 + (2.032700600098037 /
(cos( x ) / (3.951688988841374 / cos( sin( (((-1.6691938170386078
/ cos( ((2.032700600098037 / -1.3977331148909333) / cos( cos( sin(
(0.5839073910575703 + (0.5839073910575703 + (cos( 0.21588984462299265)
/ cos( 0.5839073910575703)))))))))) + 0.5839073910575703) +
(0.5839073910575703 + (cos( 0.21588984462299265) / cos(
0.5839073910575703)))))))))) + -2.1183334622298835))))))) /
-1.3977331148909333) + (cos( ((0.5839073910575703 /
(0.5839073910575703 * 0.5839073910575703)) * (sin( (x  + e^(
0.5839073910575703))) / (x  - cos( e^( 0.5839073910575703)))))) /
```

```
cos( 0.5839073910575703)))) / (3.951688988841374 / cos( sin(
(0.5839073910575703 + (0.5839073910575703 + (cos( 0.21588984462299265)
/ cos( 0.5839073910575703)))))))))) * -1.3977331148909333))) /
(sin( -4.09932580082895) / -1.3977331148909333))))) / cos( sin( (
(cos( -1.3977331148909333) / (x  * ((x  - (sin( e^( 0.5839073910575703))
/ ((0.5839073910575703 + (((cos( -1.0768808474282068) / ((
(0.5839073910575703 - e^( (sin( (x  + -2.1183334622298835)) * (sin(
(x  + 4.404029716618364)) / (x  + x ))))) / -3.9294994016950726) * cos(
cos( cos( 0.5839073910575703))))) / -1.3977331148909333) + (cos(
(e^( 0.5839073910575703) * (sin( (x  + e^( 0.5839073910575703))) / (x
- cos( e^( 0.5839073910575703)))))) / cos( 0.5839073910575703)))) /
(3.951688988841374 / cos( sin( (0.5839073910575703 + (0.5839073910575703
+ (x  / cos( 0.5839073910575703)))))))))) * -1.3977331148909333))) /
(sin( -4.09932580082895) / -1.3977331148909333))))) - e^( (sin( (cos(
(cos( (((0.5839073910575703 / (0.5839073910575703 * 0.5839073910575703))
* (sin( (x  + sin( (((2.032700600098037 / -1.3977331148909333) /
0.5839073910575703) + -2.1183334622298835)))) / (x  - cos( e^(
0.5839073910575703))))) + -1.6691938170386078)) / (((e^( x ) + sin( e^(
(x  / cos( cos( (x  / e^( 0.5839073910575703)))))))) - ((x  / cos( sin(
(((0.5839073910575703 - cos( -1.3977331148909333)) / (x  *
(3.141592653589793 * -1.3977331148909333))) / (x  / -1.3977331148909333)
)))) / -1.3977331148909333)) + ((0.5839073910575703 / ((x  / cos(
cos( x ))) - (0.5839073910575703 * cos( cos( (0.5839073910575703 / ((
(-3.9294994016950726 * sin( x )) - cos( (0.5839073910575703 -
(0.5839073910575703 * cos( (x  / 0.5839073910575703)))))) +
-2.1183334622298835))))))) - cos( -1.3977331148909333))))) + (x  + sin(
(((2.032700600098037 / -1.3977331148909333) / sin( 0.5839073910575703))
+ -2.1183334622298835))))) * sin( (cos( (x  / (cos( (x  - (((cos(
-1.0768808474282068) / (((x  - e^( (sin( (x  + (x  - ((x  - (sin( e^(
3.9169117826293363)) / ((0.5839073910575703 + (((cos( -1.0768808474282068)
/ (((3.141592653589793 - e^( (sin( (x  - (sin( (x  - 0.5839073910575703))
+ 0.5839073910575703))) * (sin( (x  + 4.404029716618364)) / (x  /
4.5366847258566025))))) - -3.9294994016950726) + cos( cos( cos( ((x  -
-2.1183334622298835) / (((-3.488290785681106 / -3.423767380013757) /
(4.496363738665307 + (2.032700600098037 + (cos( -2.9602495684565455) / (
3.951688988841374 / (x  - cos( e^( 0.5839073910575703)))))))) +
-2.1183334622298835))))))) * -1.3977331148909333) * (cos( (
(0.5839073910575703 / (0.5839073910575703 * 0.5839073910575703)) * (sin(
(-1.7908824571411053 + e^( 0.5839073910575703))) / (x  - cos( e^( x ))))))
+ e^( 0.5839073910575703)))) / (3.951688988841374 / cos( sin(
(0.5839073910575703 + (-2.8354922153845505 + (cos( 0.21588984462299265)
/ cos( 0.5839073910575703)))))))))) * -1.3977331148909333)))) * sin(
(cos( cos( 0.5839073910575703)) + (sin( 0.5839073910575703) / (x  - ((((
2.032700600098037 / -1.3977331148909333) / sin( ((sin( (cos( (x  - (x  +
e^( 0.5839073910575703)))) + sin( (x  + cos( sin( e^( (0.5839073910575703
* (sin( (x  + 4.404029716618364)) / (x  + x ))))))))) / cos( x )) / x ))
```

```
) + -2.1183334622298835) + 0.5839073910575703)))))))) /
-3.9294994016950726) * cos( 0.5839073910575703))) / -1.3977331148909333)
/ ((x  - (cos( 0.5839073910575703) + (sin( e^( 0.5839073910575703)) / (((
x  - (sin( e^( (((0.5839073910575703 / ((x  / cos( cos( x ))) -
(0.5839073910575703 * cos( (cos( 4.404029716618364) / (x  + cos(
(-1.6691938170386078 / x )))))))) - cos( (sin( (x  - cos( ((
0.5839073910575703 / (0.5839073910575703 * (sin( 0.5839073910575703) *
cos( 0.5839073910575703)))) * (0.5839073910575703 / (0.5839073910575703
* 0.5839073910575703)))))) * -1.3977331148909333))) / -4.09932580082895)))
/ (4.5366847258566025 / (3.951688988841374 / cos( -1.3977331148909333)))))
* -1.3977331148909333) - cos( e^( 0.5839073910575703)))))) / cos( cos( (x
- cos( e^( 0.5839073910575703)))))))) + 3.951688988841374))) + (sin( e^(
0.5839073910575703)) / (x  - cos( e^( 0.5839073910575703)))))))))) /
-3.9294994016950726))
```

We see in Figure 8 that we quickly get to a total error of under 4 in about 50 generations. During those 50 generations we see there was a strong selection for individuals with increased lengths. It peaks at about generation 50 and then takes a nose dive, decreasing by over 3 times. It follows this trend of sharp increases then decreases while following a steadily increasing trend, but never reaching its pinnacle at generation 50. And even though our final approximation is long, it still is about a third shorter than the best approximation at generation 50. So running the program for over 16 hours yielded a far better approximation than if we would have been satisfied at generation 30. If we would have left the program running until 600 generations had completed, could we have gotten an even better approximation with a shorter length. We'll leave this to those who have a more powerful computer and time to spare.

7 Conclusion

Although our results may seem a bit discouraging at first given the size of our approximate solution, I am not entirely convinced that it is a pointless exercise. Genetic programming, specifically used for symbolic regression, offer may offer insights into mathematical relationships that have eluded us thus far. I find it fascinating to think that we, as products of natural selection, can team up with a piece of mostly silicon rock cooked up in a star, and that by the very structure of it and by the patterns of our minds imprinted on it, invoke new discoveries about our existence. Maybe this is going too far. But we all need our inspirations to work with math or science, right?

References

[1] Koza, John. *Genetic Programming.* The MIT Press. Cambridge, 1992.

[2] Poli, Riccardo. *A Field Guide to Genetic Programming*. Lulu Enterprises. London, 2008.

[3] Burgess, G. *Finding approximate analytic solutions to differential equations using genetic programming*. Australian DOD. Sydney, 1999.

8 Appendix: The Code

8.1 Matlab - example.m

```
function yprime = example(x,y)
    yprime = cos(2*x.*y.^2);
end
```

8.2 Matlab - ToGP.m

```
%write a ode45 solution to dat file

xStart = 0;
xFinish = 5;

[x,y] = ode45(@example,[xStart:.01:xFinish],1);

TOLERANCE = 0.01;
NVAR = 1; %number of variables
NRAND = 100; %number of random constants
MIN_RAND = -5; %range of random constants
MAX_RAND = 5;
NFITCASES = length(x); %how many sample points

resSum = TOLERANCE + 1; %initialized to enter while loop
counter = 0;

while resSum > TOLERANCE && counter < NFITCASES
  resSum = 0;
  counter = counter + 1;
  p = polyfit(x,y,counter);
  yp = polyval(p,x);
  Residual = abs(y-yp);
  resSum = sum(Residual); %calcs the error between polynomial approx and actual
end

%print normal regression values
counter
```

```
p
resSum

plot(x,y,'k')
title('yprime = cos(2*x*y^2), y(0)=1')
xlabel('x')
ylabel('y')
%legend('Numerical Solution','Polyfit Approximation')

%write to file
fid1 = fopen('C:NetBeansProjects\TinyGP\example.dat','w');

fprintf(fid1,'%i %i %i %i %i %i\n', NVAR,NRAND,MIN_RAND,MAX_RAND,NFITCASES, TOLERANCE);
fprintf(fid2,'%i %i %i %i %i %i\n', NVAR,NRAND,MIN_RAND,MAX_RAND,NFITCASES, TOLERANCE);

%write data points to file
for k = 1 : NFITCASES
    fprintf(fid1, '%f %f\n', x(k), y(k));
    fprintf(fid2, '%f %f\n', x(k), y(k));
end

disp('Successfully written to file');
```

8.3 Java: TinyGP.java

```
/*
 * Program:   TinyGP.java
 * Adapted from Tiny GP by Riccardo Poli
 * http://cswww.essex.ac.uk/staff/rpoli/TinyGP/tiny_gp.java
 */
package tinygp;

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

public class TinyGP
{     //FIELD of ATTRIBUTES
      //Parameters
      private static final int
          MAX_LEN = 1000,
          POPSIZE = 10000,
          DEPTH   = 5,      //depth of tree with root node = 0
          GENERATIONS = 600,
          TSIZE = 2;        //number of players per tournament
      private static final double
          PMUT_PER_NODE  = 0.3, //prob of mutation for each node
```

```
    CROSSOVER_PROB = 0.8, //prob of crossover
    PENALTY_COEFFICIENT = 0.0001; //penalty if length is too long
private double tolerance;

//Interpreter for functions
private static final int
    FSET_START = 110,
    FSET2_START = FSET_START,
    ADD = FSET2_START,        //110
    SUB = FSET2_START + 1,    //111
    MUL = FSET2_START + 2,    //112
    DIV = FSET2_START + 3,    //113
    FSET2_END = DIV,          //113
    FSET1_START = FSET2_END + 1, //114
    SIN = FSET1_START,        //114
    COS = FSET1_START + 1,    //115
    EXP = FSET1_START + 2,    //116
    FSET1_END = EXP,          //115
    FSET_END = FSET1_END;     //115

//Nodes
private static int numOfVar, fitnessCases, numOfRand;
private static double [] x = new double[FSET_START]; //x[0]=x, x[1-109]=const
private static Random rd = new Random();
private static long seed;
private static double randomMin, randomMax;  //determines range of random

//Programs
private static double [][] targets, result;   //the domain set (x,y)
private char [][] pop;//array of population trees
private static char []
    program,
    buffer = new char[MAX_LEN];
private static int progCount;
private static int genCount;
private int seedCount = 0;
private double [] fitness;        //array of error summations

BufferedWriter output;
String eq = "";

//Assessment
private int best; //index of best individual
private static double
    bestInPop = 0.0,      //the fitness of best individual
    aveOfPop = 0.0,       //the average of all individuals in pop
```

```
    avgLen;               //the average length of all individuals
private long cpuTime;

//CONSTRUCTOR
public TinyGP( String fname, long s )
{
      fitness =  new double[POPSIZE];
      seed = s;

      if ( seed >= 0 )
          rd.setSeed(seed);

      setupFitness(fname);

      openFileToWrite("TerminalValues.txt");

      x[0] = 0;
      x[1] = Math.PI;
      x[2] = -Math.PI;
      x[3] = Math.PI/2;
      x[4] = -Math.PI/2;

      for ( int i = 5; i < FSET_START; i ++ )
      {
          x[i] = (randomMax - randomMin) * rd.nextDouble() + randomMin;

          try { output.write(Double.toString(x[i]) + "\n");}
          catch (IOException ex) {}
      }

      closeFile();

      pop = createPop(POPSIZE, DEPTH, fitness );
}

public void setupFitness(String fname)
{
      try
      {
        BufferedReader in = new BufferedReader(new FileReader(fname));
        String line = in.readLine();
        StringTokenizer tokens = new StringTokenizer(line);

        numOfVar = Integer.parseInt(tokens.nextToken().trim());
        numOfRand = Integer.parseInt(tokens.nextToken().trim());
        randomMin = Double.parseDouble(tokens.nextToken().trim());
```

```
        randomMax =  Double.parseDouble(tokens.nextToken().trim());
        fitnessCases = Integer.parseInt(tokens.nextToken().trim());
        tolerance = Double.parseDouble(tokens.nextToken().trim());
        targets = new double[fitnessCases][numOfVar+1];
        result = new double[fitnessCases][2];

        if (numOfVar + numOfRand >= FSET_START )
          System.out.println("too many variables and constants");

        for (int i = 0; i < fitnessCases; i ++ )
        {
          line = in.readLine();
          tokens = new StringTokenizer(line);

          for (int j = 0; j <= numOfVar; j++)
          {
            targets[i][j] = Double.parseDouble(tokens.nextToken().trim());
          }

          result[i][0] = targets[i][0]; //same x values
        }

        in.close();
      }
      catch(FileNotFoundException e) {
        System.out.println("ERROR: Please provide a data file");
        System.exit(0);
      }
      catch(Exception e ) {
        System.out.println("ERROR: Incorrect data format");
        System.exit(0);
      }
}

public char [][] createPop(int popSize, int depth, double [] fitness )
{
      char [][]localPop = new char[popSize][];

      //get the seeded pop first

      for (int i = seedCount; i < popSize; i ++ )
      {
        localPop[i] = createRandomIndiv( depth );
        fitness[i] = fitnessFunction( localPop[i] );
      }
      return( localPop );
```

```
}

public char [] createRandomIndiv( int depth )
{
      char [] ind;
      int len;

      len = grow( buffer, 0, MAX_LEN, depth );

      while (len < 0 )
        len = grow( buffer, 0, MAX_LEN, depth );

      ind = new char[len];

      System.arraycopy( buffer, 0, ind, 0, len );
      return( ind );
}

//returns length of local program array
public int grow( char [] buffer, int pos, int max, int depth )
{
      char prim;
      int oneChild;

      if ( pos >= max )
        return( -1 );

      if ( pos == 0 )
        prim = 1; //first node must be a function
      else
        prim = (char) rd.nextInt(2); //if prim is 0-->node, 1-->function

      if ( prim == 0 || depth == 0 ) //when depth max is reached, only nodes
      {
        prim = (char) rd.nextInt(numOfVar + numOfRand);
        buffer[pos] = prim;
        return(pos+1);
      }
      else
      {
        prim = (char)(rd.nextInt(FSET_END - FSET_START + 1) + FSET_START);
        switch(prim)
        {
            case SIN:
            case COS:
            case EXP:
```

```
              buffer[pos] = prim;
              return( grow( buffer, pos+1, max,depth-1) );
            case ADD:
            case SUB:
            case MUL:
            case DIV:
              buffer[pos] = prim;
              oneChild = grow( buffer, pos+1, max,depth-1);

              if ( oneChild < 0 )
                  return( -1 );

              return( grow( buffer, oneChild, max,depth-1 ) );
        }
      }
      return( 0 ); // should never get here
}

public double fitnessFunction( char [] prog )
{
      int len;
      double fit = 0.0;

      len = traverse( prog, 0 );

      if( len > 300)
          fit += (PENALTY_COEFFICIENT * (len-300));

      for (int i = 0; i < fitnessCases; i ++ )
      {
        for (int j = 0; j < numOfVar; j ++ )
            x[j] = targets[i][j];

        program = prog; //
        progCount = 0; //reset counter
        result[i][numOfVar] = run();

        fit += Math.abs( result[i][numOfVar] - targets[i][numOfVar] );
      }
      return(-fit);
}

// INTERPRETER
public double run()
{
      char primitive = program[progCount++];
```

```
      if ( primitive < FSET_START )
        return(x[primitive]);
      else
      {
        switch ( primitive )
        {
            case ADD : return( run() + run() );
            case SUB : return( run() - run() );
            case MUL : return( run() * run() );
            case SIN : return( Math.sin( run() ) );
            case COS : return( Math.cos( run() ));
            case EXP :
            {
              double argument = run();
              if ( argument >  5 )
                return( Math.exp(5) );
              else
                return( Math.exp(argument) );
            }
            case DIV :
            {
              double num = run(), den = run();

              if ( Math.abs( den ) <= 0.001 )
                return( num );
              else
                return( num / den );
            }
        }
        return( 0.0 ); // should never get here
      }
}

public int traverse( char [] buffer, int buffercount )
{
      if ( buffer[buffercount] < FSET_START )
        return( ++buffercount );

      switch(buffer[buffercount])
      {
          case SIN:
          case COS:
          case EXP:
            return( traverse( buffer, ++buffercount ) );
          case ADD:
```

```
            case SUB:
            case MUL:
            case DIV:
            return( traverse( buffer, traverse( buffer, ++buffercount ) ) );
        }

        return( 0 ); // should never get here
    }

    public void evolve()
    {
        int indivs, offspring, parent1, parent2, parent;
        double newfit;
        char []newind;
        long start = 0;

        printParams();

        stats( fitness, pop, 0 );

        for (genCount = 1; genCount <= GENERATIONS; genCount++ )
        {
          start = System.currentTimeMillis();

          if ( bestInPop > -tolerance )
          {
              System.out.print("PROBLEM SOLVED\n");

              if(output != null)
              {  try { output.close();}
                 catch(IOException ex){System.out.print("***Error closing***\n");}
              }
              System.exit( 0 );
          }
          for ( indivs = 0; indivs < POPSIZE; indivs ++ )
          {
              if ( rd.nextDouble() < CROSSOVER_PROB  )
              {
                parent1 = tournament( fitness, TSIZE );
                parent2 = tournament( fitness, TSIZE );
                newind = crossover( pop[parent1],pop[parent2] );
              }
              else
              {
                parent = tournament( fitness, TSIZE );
                newind = mutation( pop[parent], PMUT_PER_NODE );
```

```
            }

            newfit = fitnessFunction( newind );
            offspring = negativeTournament( fitness, TSIZE );
            pop[offspring] = newind;
            fitness[offspring] = newfit;
          }

          cpuTime = System.currentTimeMillis() - start;
          stats( fitness, pop, genCount );
        }

        System.out.print("PROBLEM *NOT* SOLVED\n");

        if(output != null)
        {  try { output.close();}
           catch(IOException ex){System.out.print("***Error closing***\n");}
        }
        System.exit( 1 );
}

public void printParams()
{
     System.out.print("-- TINY GP (Java version) --\n");
     System.out.print("SEED="+seed+"\nMAX LEN="+MAX_LEN+
              "\nPOPSIZE="+POPSIZE+"\nDEPTH="+DEPTH+
              "\nCROSSOVER PROB="+CROSSOVER_PROB+
              "\nPROB MUT PER NODE="+PMUT_PER_NODE+
              "\nRANDOM MIN="+randomMin+
              "\nRANDOM MAX="+randomMax+
              "\nGENERATIONS="+GENERATIONS+
              "\nTOURNY SIZE="+TSIZE+"\nTOLERANCE="+tolerance+
              "\n-----------------------------------\n");
}

public void stats( double [] fitness, char [][] pop, int gen )
{
        best = rd.nextInt(POPSIZE);
        int nodeCount = 0;
        bestInPop = fitness[best];
        aveOfPop = 0.0;
        String fileName = "ListOfEquations.txt";
        String fileConc = "Generation";


        for (int i = 0; i < POPSIZE; i ++ )
```

```
{ eq = ""; //to reset equation;
  nodeCount +=  traverse( pop[i], 0 );
  aveOfPop += fitness[i];

  if ( fitness[i] > bestInPop )
  {
    best = i;
    bestInPop = fitness[i];
  }
}
avgLen = (double) nodeCount / POPSIZE;
aveOfPop /= POPSIZE;

printIndiv( pop[best], 0 );

System.out.print("\nGeneration="+gen+" Best Fitness="+(-bestInPop)
        +" Avg Fitness="+(-aveOfPop)+" Avg Size="+avgLen+
        "\nBest Individual: ");
System.out.println(eq);
System.out.flush();

//Print generation, fitness of best, average fitness, average length
openFileToWrite("Evaluations.txt");
try {
    output.write(
        String.format("%5d  %10.4f  %15.4f  %12.4f %5d\n",gen,
        (-bestInPop),(-aveOfPop),avgLen,cpuTime));
    output.flush();
}catch (IOException ex){ System.out.print("***Error***\n");}
closeFile();

if( genCount % 25 == 0 || -bestInPop < tolerance)
{   openFileToWrite(fileName);

    try {
        output.write(String.format("Generation: %d Error: %.4f\n" +
                " f(x) = %s\n\n", genCount, (-bestInPop), eq));
        output.flush();
    }catch (IOException ex){ System.out.print("***Error***\n");}
    closeFile();
}

if( genCount % 25 == 0 || -bestInPop < tolerance)
{
    fitnessFunction( pop[ best ] );
```

```
            if( -bestInPop < tolerance )
                fileConc = "BEST_SOLUTION_ARRAY.txt";
            else
                fileConc = fileConc.concat(
                        Integer.toString(genCount)+ ".txt");

            openFileToWrite( fileConc );

            for( int i = 0; i < fitnessCases; i++){
                try {
                   String x1 = Double.toString(result[i][0]);
                   String y1 = Double.toString(result[i][numOfVar]);
                   output.write(x1+" "+y1+"\n");
                   output.flush();
                } catch(IOException ex){}//write data points
              }

             if(output != null)
             {   try { output.close();}
                 catch(IOException ex){System.out.print("***Error closing***\n");}
             }
      }
}

public int tournament( double [] fitness, int tSize )
{
      best = rd.nextInt(POPSIZE);
      int competitor;
      double  fbest = -1.0e34;

      for (int i = 0; i < tSize; i ++ )
      {
        competitor = rd.nextInt(POPSIZE);

        if ( fitness[competitor] > fbest )
        {
          fbest = fitness[competitor];
          best = competitor;
        }
      }
      return( best );
}

public char [] crossover( char []parent1, char [] parent2 )
{
      int xo1Start, xo1End, xo2Start, xo2End;
```

```
        char [] offspring;
        int len1 = traverse( parent1, 0 );
        int len2 = traverse( parent2, 0 );
        int lenoff;

        xo1Start =  rd.nextInt(len1);
        xo1End = traverse( parent1, xo1Start );

        xo2Start =  rd.nextInt(len2);
        xo2End = traverse( parent2, xo2Start );

        lenoff = xo1Start + (xo2End - xo2Start) + (len1-xo1End);

        offspring = new char[lenoff];

        System.arraycopy( parent1, 0, offspring, 0, xo1Start );
        System.arraycopy( parent2, xo2Start, offspring, xo1Start,
                     (xo2End - xo2Start) );
        System.arraycopy( parent1, xo1End, offspring,
                     xo1Start + (xo2End - xo2Start),
                     (len1-xo1End) );

        return( offspring );
}

public char [] mutation( char [] parent, double pmut )
{
        int len = traverse( parent, 0 );
        int mutsite;
        char [] parentcopy = new char [len];

        System.arraycopy( parent, 0, parentcopy, 0, len );

        for (int i = 0; i < len; i ++ )
        {
          if ( rd.nextDouble() < pmut )
          {
              mutsite =  i;

              if ( parentcopy[mutsite] < FSET_START )
                parentcopy[mutsite] = (char) rd.nextInt(numOfVar+numOfRand);
              else
              {
                switch(parentcopy[mutsite])
                {
                    case SIN:
```

```
                    case COS:
                    case EXP:
                        parentcopy[mutsite] =
                            (char) (rd.nextInt(FSET1_END - FSET1_START + 1)
                                + FSET1_START);
                        break;
                    case ADD:
                    case SUB:
                    case MUL:
                    case DIV:
                       parentcopy[mutsite] =
                            (char) (rd.nextInt(FSET2_END - FSET2_START + 1)
                                 + FSET2_START);
                       break;
                  }
               }
            }
      }
      return( parentcopy );
}

public int negativeTournament( double [] fitness, int tsize )
{
      int worst = rd.nextInt(POPSIZE), i, competitor;
      double fworst = 1e34;

      for ( i = 0; i < tsize; i ++ )
      {
        competitor = rd.nextInt(POPSIZE);

        if ( fitness[competitor] < fworst )
        {
          fworst = fitness[competitor];
          worst = competitor;
        }
      }
      return( worst );
}

public int printIndiv( char []buffer, int buffercounter )
{
      int a1=0, a2;

      if ( buffer[buffercounter] < FSET_START )
      {
          if ( buffer[buffercounter] < numOfVar )
```

```
            eq = eq.concat( "x " );//+ (buffer[buffercounter] + 1 )+ " "
        else
            eq = eq.concat( Double.toString(x[buffer[buffercounter]]));
            return( ++buffercounter );
    }
    else
    {
        switch(buffer[buffercounter])
        {
            case SIN: eq = eq.concat( "sin( ");
                      a1= ++buffercounter;
                      break;
            case COS: eq = eq.concat( "cos( ");
                      a1 = ++buffercounter;
                      break;
            case EXP: eq = eq.concat( "e^( ");
                      a1= ++buffercounter;
                      break;
            case ADD: eq = eq.concat( "(");
                      a1=printIndiv( buffer, ++buffercounter );
                      eq = eq.concat( " + ");
                      break;
            case SUB: eq = eq.concat( "(");
                      a1=printIndiv( buffer, ++buffercounter );
                      eq = eq.concat( " - ");
                      break;
            case MUL: eq = eq.concat( "(");
                      a1=printIndiv( buffer, ++buffercounter );
                      eq = eq.concat( " * ");
                      break;
            case DIV: eq = eq.concat( "(");
                      a1=printIndiv( buffer, ++buffercounter );
                      eq = eq.concat( " / ");
                      break;
        }

        a2=printIndiv( buffer, a1 );

        eq = eq.concat( ")");
        return( a2);
    }
}

public void seedIndividuals()
{
```

```
            //read in serialized individuals, increment seedCount, get fitness
            //handle with try...catch, if no file or no seeds, catch and continue
        }

        public void openFileToWrite(String fileName)
        {
            try{
                output = new BufferedWriter(new FileWriter(fileName, true));
            }catch(IOException ioex)
            {
                System.out.println("\n***Error opening file to write***\n\n");
            }
        }

        public void closeFile()
        {
            if( output!= null){
                try {
                      output.close();
                } catch (IOException ex) {
                      System.out.print("\n***Error closing file***\n");
                }
            }
        }

        public static void main(String[] args) {
              String fname = "example.dat";
              long s = -1; //seed needs to be the same for seeding population

              TinyGP gp = new TinyGP(fname, s);
              gp.evolve();
        }
}
```

8.4 Matlab: FromGP.m

```
%retrieve data from gp
approx = ones(501,9);

a = load('C:NetBeansProjects\TinyGPwTRIG\Generation0.txt');
b = load('C:NetBeansProjects\TinyGPwTRIG\Generation25.txt');
c = load('C:NetBeansProjects\TinyGPwTRIG\Generation50.txt');
d = load('C:NetBeansProjects\TinyGPwTRIG\Generation150.txt');
e = load('C:NetBeansProjects\TinyGPwTRIG\Generation375.txt');
%f = load('C:NetBeansProjects\TinyGPwTRIG\Generation125.txt');
%g = load('C:NetBeansProjects\TinyGPwTRIG\Generation150.txt');
```

```
%h = load('C:NetBeansProjects\TinyGPwTRIG\Generation175.txt');

approx(:,1)=a(:,1);
approx(:,2)=a(:,2);
approx(:,3)=b(:,2);
approx(:,4)=c(:,2);
approx(:,5)=d(:,2);
approx(:,6)=e(:,2);
%approx(:,7)=f(:,2);
%approx(:,8)=g(:,2);
%approx(:,9)=h(:,2);


plot(approx(:,1),approx(:,2),'g')
hold on
plot(approx(:,1),approx(:,3),'r')
hold on
plot(approx(:,1),approx(:,4),'c')
hold on
plot(approx(:,1),approx(:,5),'m')
hold on
%plot(approx(:,1),approx(:,6),'b')
%hold on
%plot(approx(:,1),approx(:,7),'k--')
%hold on
%plot(approx(:,1),approx(:,8),'k:')
%hold on
%plot(approx(:,1),approx(:,9),'k-.')
xlabel('x')
ylabel('y')
title('yprime = cos(2*x*y^2)')
legend('Numerical','Gen 0', 'Gen 25', 'Gen 50', 'Gen 150')

figure
plot(x,y,approx(:,1),approx(:,5),'g')
hold on
plot(approx(:,1),approx(:,6),'r--')
xlabel('x')
ylabel('y')
title('Best Fit & Numerical Solution')
legend('Numerical','Gen 150','Gen 375')
```

8.5 Matlab: GPEval.m

```
eval = ones(386, 5);
```

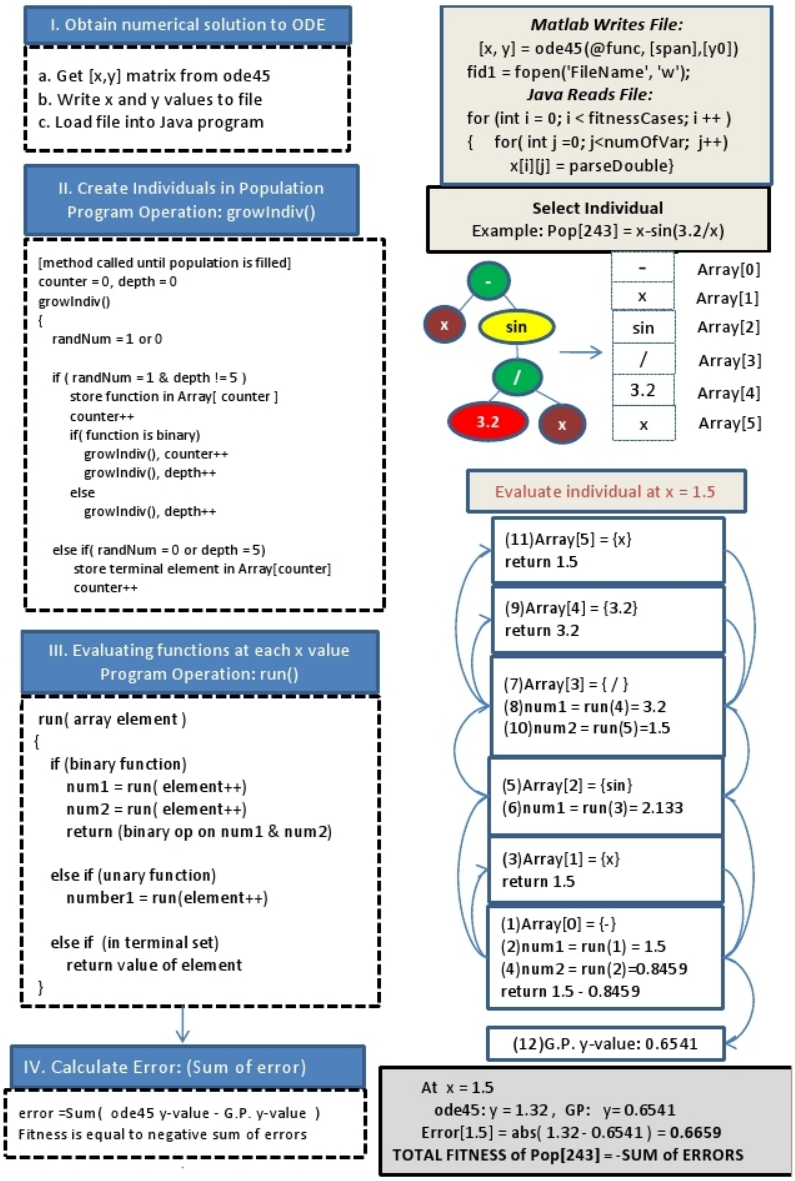
```
eval = load('C:NetBeansProjects\TinyGP\Evaluations.txt');

figure
plot(eval(:,1),eval(:,2),'r')
xlabel('Generations')
ylabel('Best Individual Fitness')
figure

plot(eval(:,1),eval(:,3),'k')
xlabel('Generations')
ylabel('Average Population Fitness')
figure

plot(eval(:,1),eval(:,4))
xlabel('Generations')
ylabel('Average length of individuals')
figure

plot(eval(:,1),eval(:,5),'r')
xlabel('Generation')
ylabel('CPU ticks/Generation')
```


I. Obtain numerical solution to ODE
a. Get [x,y] matrix from ode45
b. Write x and y values to file
c. Load file into Java program
Matlab Writes File:
[x, y] = ode45(@func, [span],[y0])
fid1 = fopen('FileName', 'w');
Java Reads File:
for (int i = 0; i < fitnessCases; i ++)
{ for(int j =0; j<numOfVar; j++)
x[i][j] = parseDouble}
II. Create Individuals in Population
Program Operation: growIndiv()
[method called until population is filled]
counter = 0, depth = 0
growIndiv()
{
randNum = 1 or 0
if (randNum = 1 & depth != 5)
store function in Array[counter]
counter++
if(function is binary)
growIndiv(), counter++
growIndiv(), depth++
else
growIndiv(), depth++
else if(randNum = 0 or depth = 5)
store terminal element in Array[counter]
counter++
Select Individual
Example: Pop[243] = x-sin(3.2/x)
-
x
sin
/
3.2
x
Array[0]
Array[1]
Array[2]
Array[3]
Array[4]
Array[5]
Evaluate individual at x = 1.5
(11)Array[5] = {x}
return 1.5
(9)Array[4] = {3.2}
return 3.2
(7)Array[3] = { / }
(8)num1 = run(4)= 3.2
(10)num2 = run(5)=1.5
(5)Array[2] = {sin}
(6)num1 = run(3)= 2.133
(3)Array[1] = {x}
return 1.5
(1)Array[0] = {-}
(2)num1 = run(1) = 1.5
(4)num2 = run(2)=0.8459
return 1.5 - 0.8459
(12)G.P. y-value: 0.6541
III. Evaluating functions at each x value
Program Operation: run()
run(array element)
{
if (binary function)
num1 = run(element++)
num2 = run(element++)
return (binary op on num1 & num2)
else if (unary function)
number1 = run(element++)
else if (in terminal set)
return value of element
}
IV. Calculate Error: (Sum of error)
error =Sum(ode45 y-value - G.P. y-value)
Fitness is equal to negative sum of errors
At x = 1.5
ode45: y = 1.32 , GP: y= 0.6541
Error[1.5] = abs(1.32 - 0.6541) = 0.6659
TOTAL FITNESS of Pop[243] = -SUM of ERRORS
V. Perform Reproductive Operations
randomly select for mutation or crossover
if (mutation)
a. at each node, calculate if mutate
b. return child
else if(crossover)
a. Randomly select two individuals
b. Individual with higher fitness is Parent1
c. Randomly select two individuals
d. Individual with higher fitness is Parent2
Random Parent1
Random Parent2
e
sin
+
2
x
-
x
sin
/
3
x
Crossover
VI. Repeat III-VI until Tolerance is Met
while(no indiv's fitness < Tol or not MaxGen)
{
Calculate fitness of each individual
Perform reproduction
}