Genetic Programming

This article presents a method for optimizing expressions to solve a given problem using the strategy of Darwinism. In contrast to genetic algorithms, which evolve an encoded representation of the solution, genetic programming evolves the solution expression directly. The *Mathematica* implementation makes use of the built-in features of functional programming, recursion, and hierarchical data structures. An application to symbolic regression is presented.

Robert B. Nachbar

Evolutionary computing has become a popular method to robustly search very large solution spaces for optimal or near-optimal solutions to a wide variety of problems. These methods begin with an initial, usually randomly generated, population of individual instances of a data structure. Each element of the population can be evaluated to a solution for the problem at hand. The term 'solution' is used rather loosely here because these primordial individuals are generally not very good solutions. Then, in the spirit of survival of the fittest, they are propagated directly, through recombination with mates, or by mutation into new, and hopefully more fit, individuals. This iterative process is repeated until either one grows weary or a satisfactory solution is found. The governing principle of these stochastic methods is that the fittest individuals possess a part of the optimal solution. By operating on them to make more-fit individuals one will eventually discover the optimal solution. Unlike the more traditional analytic methods (such as least squares or conjugate gradients), these methods can avoid local minima and sample a large portion of the solution space because of the stochastic nature of the multiple search trajectories.

John R. Koza has recently described his approach to genetic programming with the Lisp language [Koza 1992]. It is difficult to provide a more thorough introduction to this topic than his, and the reader is directed to that work for further examples and more detailed discussions. What I intend to show here is that genetic programming is easily done in *Mathematica* as well.

Evolutionary Computing

In a genetic algorithm (GA), the individuals are strings of characters that encode the parameters that specify a solution for the problem at hand. A close analogy between the strings and chromosomes has been drawn [Goldberg 1989]. In a GA, mutation is achieved by randomly selecting one of the letters of a string and exchanging it for a different letter of the alphabet. Typically, a binary alphabet is used (0 and 1). Recombination via mating is carried out by crossing over part of one parent with the homologous part of the other parent. This is done by selecting a point at random along the string and then exchanging the distal portions between the parents, thus generating two new offspring. Because the string is a coded representation of a solution, its length is fixed. In spite of this rigid linear data structure, GAs have been applied successfully to a large number of problems. See [Goldberg 1989] and [Davis 1991] for further details and examples. James Freeman has described the use of *Mathematica* for GAs [Freeman 1993].

A genetic program (GP) shares most of the features of a GA. However, the data structure that GP uses is hierarchical rather than linear. In addition, instead of employing an alphabet to encode the solution parameters, the parameters themselves are stored in the data structure along with the functions that operate on them. A graph-theoretical tree is a suitable hierarchical data structure for a GP. (This is the same data structure that most compilers use to parse mathematical expressions.) The parameters occupy the terminal or external nodes (sometimes called leaves) and the functions occupy the internal nodes (the highest of which is known as the root). In an expression tree, a subtree is the analog of a substring in a genetic algorithm. The genetic operations of mutation and crossover are applied to the subtrees.

Expression trees are customarily drawn upside down, with the root at the top and the leaves at the bottom. For example, the expression $2\sin x + bx + c$ can be represented as the tree shown in Figure 1. This expression tree is in fact a reusable program whose inputs are the symbolic terminal nodes and whose output is the result produced by the root node.

I first became interested in genetic programming for solving symbolic regression problems, a generalization of curve

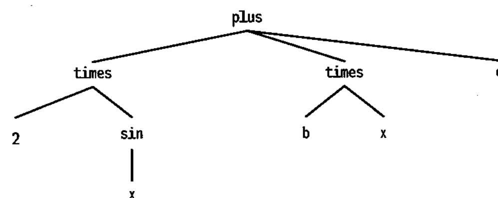


FIGURE 1. The hierarchical tree structure of the expression $2 \sin x + b x + c$.

Robert B. Nachbar is a Research Fellow in the Molecular Design and Diversity department at Merck Research Laboratories, where he works on all aspects of molecular modeling, from software development to computer-aided drug design. He received his Ph.D. in organic chemistry from Brown University in 1979.

fitting. In traditional methods, such as least squares, one uses a presumed functional form (such as $y = mx + b$) and determines only the coefficients (m and b) that best fit the observed data. In symbolic regression, one is interested in finding the best functional form as well as the coefficients. For example, given the sequence 5, 31, 121, 341, 781, 1555, 2801, 4681, what is the next entry? What is the 12th? To answer these questions, we must discover the relationship between i, the position in the sequence, and S_i, the i-th entry. The task of genetic programming is to find the appropriate combination of functions, variables, and coefficients to exactly reproduce this sequence. Then we can use the result to calculate any element of the sequence.

Expression Trees

Functions and Terminals

The internal representation of expressions in *Mathematica* (as revealed with the FullForm function) is similar to the heirarchical tree structure shown in Figure 1.

```
In[1]:= FullForm[(x+1)(y-z)]
Out[1]//FullForm= Times[Plus[1, x], Plus[y, Times[-1, z]]]
```

A powerful feature of the tree structure is that it is recursive. Each subtree is itself a valid tree. All the internal nodes (Times, Plus, Plus, and Times in this instance) are at the roots of subtrees. Even the terminal nodes (1, x, y, -1, and z) are themselves trivial expression trees. From this example, it is obvious that functions occupy internal nodes, and numbers and variables occupy terminal nodes.

Evaluation of an expression tree is carried out recursively. Since the internal nodes are built-in functions, we do not need a special evaluation function; *Mathematica* will do it automatically. For a good description of trees and recursive programming, see [Gaylord et al. 1993].

Notice that the expression above has been rearranged by *Mathematica* into a canonical form equivalent to (1 + x) * (y + (-1 * z)). This rearrangement is relatively minor and could be ignored. However, Times and Divide present difficulties because their use can introduce a new function, namely Power.

```
In[2]:= Times[x, x] // FullForm
Out[2]//FullForm= Power[x, 2]

In[3]:= Divide[x, y] // FullForm
Out[3]//FullForm= Times[x, Power[y, -1]]
```

The appearance of Power may be undesirable for the problem at hand because genetic recombination of individuals could easily replace the second argument of Power with an expression that does not evaluate to an integer. In addition, Power may not be a member of the set of allowed functions. Therefore, by letting *Mathematica* automatically evaluate the expression tree, we lose some control over the function set.

Another difficulty arises when we use built-in functions whose arguments are made up entirely of numbers. We loose the full tree structure and are left with a single node.

```
In[4]:= Times[5, Plus[3, 2]] // FullForm
Out[4]//FullForm= 25
```

One way of overcoming these problems is to keep the functions and their arguments in lists, such as {Times, {5}, {Plus, {3}, {2}}} (see [Gaylord et al. 1993]). Alternatively, we can introduce our own arithmetic functions (plus, times, subtract, and divide) that do not evalute to anything. Both methods preserve the full structure of the expression and both require a function that can evaluate the expression and return a usable result. The evaluator for the former representation must recursively traverse the nested lists while it builds up a *Mathematica* expression. For the latter method, the evaluator need only provide a set of replacement rules. We have found the second approach more convenient, and it probably requires less memory as well.

```
In[5]:= evalRules = {plus -> Plus, times -> Times,
          subtract -> Subtract, divide -> Divide} ;
        Eval[expr_] := expr /. evalRules
```

Thus, for the examples above:

```
In[7]:= times[plus[x, 1], subtract[y, z]]
Out[7]= times[plus[x, 1], subtract[y, z]]

In[8]:= Eval[%]
Out[8]= (1 + x) (y - z)

In[9]:= times[x, x]
Out[9]= times[x, x]

In[10]:= Eval[%]
         2
Out[10]= x

In[11]:= divide[x, y]
Out[11]= divide[x, y]

In[12]:= Eval[%]
         x
Out[12]= -
         y

In[13]:= times[5, plus[3, 2]]
Out[13]= times[5, plus[3, 2]]

In[14]:= Eval[%]
Out[14]= 25
```

At this point, it is appropriate to discuss two requirements of the sets of functions and terminals used in GP: *closure* and *sufficiency*. The value at any terminal node, or the result of any function at an internal node, can be an argument to the function at the next level up the tree. One must therefore ensure that the choices of terminals and functions are fully compatible so that functions can accept any arguments they may receive and still yield valid results. That is, the functions should be well defined and *closed*. For example, one would not want to mix numerical and Boolean terminals, as the result is still partially unevaluated:

```
In[15]:= Eval[ plus[2, True, 3] ] // FullForm
Out[15]//FullForm= Plus[5, True]
```

Nor does one want to encounter undefined results:

```
In[16]:= Eval[ divide[subtract[x, x], 0] ] // FullForm
         General::dbyz: Division by zero.
         Infinity::indet:
            Indeterminate expression 0 ComplexInfinity
              encountered.
Out[16]//FullForm= Indeterminate
```

The problem of division by zero is easily remedied by using *protected division*, which tests the denominator before the division is carried out. If the denominator is zero, a value such as 1 can be returned, which maintains closure [Koza 1992, 82]. However, it is more practical to return a very large machine number, which is closer to the true (infinite) value and so is more appropriate in real-world problems [Lee 1994]. (We could allow 1/0 to evaluate to Infinity and Infinity/Infinity to evalute to Infinity, but that would cause problems further down the road if we try to compile the result.) Similar protected functions should be used for Sqrt and Log, and Exp should be protected from over- and underflow.

```
In[17]:= bigInteger = 2^63 - 1 ;
         bigReal = N[bigInteger] ;
         PDivide[n_Integer, 0] := bigInteger
         PDivide[n_Integer, 0.] := bigInteger
         PDivide[n_, 0] := bigInteger
         PDivide[n_Real, 0] := bigReal
         PDivide[n_Real, 0.] := bigReal
         PDivide[n_, 0.] := bigReal
         PDivide[n_, d_?NumberQ] := Divide[n, d]

In[26]:= evalRules = evalRules /. Divide -> PDivide
Out[26]= {plus -> Plus, times -> Times, subtract -> Subtract,
           divide -> PDivide}

In[27]:= Eval[ divide[subtract[x, x], 0] ]
Out[27]= 9223372036854775807
```

All of the type-testing for PDivide is necessary so that actual division does not take place before it is known that the denominator is nonzero and that an appropriate precision result is returned.

```
In[28]:= Eval[ divide[subtract[x, 1], y] ]
Out[28]= PDivide[-1 + x, y]
```

Sufficiency of the function set and terminal set means that some combination of their elements is capable of producing an expression that is the solution to the problem. Ensuring sufficiency is the responsibility of the user and is not always a straightforward task. If variables and functions without sufficient explanatory power are used, the solution cannot be found. For example, if the data shows periodic behavior, but periodic functions (such as Sin or Mod) are not included in the function set, an adequate solution cannot be constructed. One must also be careful not to include extraneous functions or terminals as this can hamper the performance of the search for the solution.

Conditional Functions

Logical tests are very common in procedural programs written in languages such as FORTRAN or C, but somewhat unexpected in mathematical expressions. The If function in *Mathematica* not only controls which branch is followed, but it also returns the final value of the branch followed. Because the first argument of If must be a Boolean, we cannot use it directly. The functionality we seek can be implemented with a modified If function that uses only numerical arguments. For example:

```
In[29]:= Attributes[ltz] = {HoldRest} ;
         AppendTo[evalRules,
            ltz[test_, t_, f_] :> If[test < 0, t, f]] ;

In[31]:= Eval[ ltz[-3, x, y] ]
Out[31]= x

In[32]:= Eval[ ltz[2, x, y] ]
Out[32]= y
```

The attribute HoldRest is necessary to prevent *Mathematica* from evaluating t and f prior to comparing test against zero. The reason is not obvious from this example, but if either t or f produced a side effect that was used elsewhere, we would want only the correct side effect to be produced. (The reader might try clearing the attribute and including a Print in each of the branches to see the effect.)

Constant Terminals

Frequently we need constants to describe fully the solution to a problem. Even if we do not include them in the terminal set, they can be constructed during the evolution of the run. Integer and rational constants are the easiest to come across.

```
In[33]:= Eval[ plus[x, x] ]
Out[33]= 2 x
```

Irrational constants can also be spontaneously generated. In a GP used to find trigonometric identities [Koza 1992, 242–245], the constant $\pi/2$ was approximately constructed as

```
In[34]:= Eval[ Subtract[2,
            Sin[Sin[Sin[Sin[Sin[Sin[Times[Sin[Sin[1]],
               Sin[Sin[1]]]]]]]]]] ] // N
Out[34]= 1.56721

In[35]:= Pi/2 // N
Out[35]= 1.5708
```

To facilitate the incorporation of constants, the *ephemeral random constant* is introduced. Whenever it is encountered, a random number of the appropriate type is generated. Random constants are used as terminals during the creation of the initial generation. During the evolution of the population, they can be recombined in many ways to form new constants.

```
In[36]:= ephemeralReal := Random[Real, {-1, 1}]

In[37]:= Table[ephemeralReal, {5}]
Out[37]= {-0.894461, -0.84196, -0.367542, 0.300289, 0.0639713}
```

Algebraic Simplification

As mentioned above, we chose to define our own arithmetic functions so that we can maintain better control of them. With this change we also lost all of *Mathematica*'s built-in simplification. For example,

```
In[38]:= Plus[Subtract[x, x], Times[2, Divide[Plus[x, 2], 2]]]
Out[38]= 2 + x

In[39]:= plus[subtract[x, x], times[2, divide[plus[x, 2], 2]]]
Out[39]= plus[subtract[x, x], times[2, divide[plus[x, 2], 2]]]
```

Just how much simplification is *necessary* is not always clear. The result 2 + x is certainly concise, which is desirable for a solution to a regression problem. However, we have also lost a great deal of genetic flexibility. For example, there are only three positions available at which genetic operations may occur in the simplified result, compared with 11 in the original expression.

We hope to strike a useful balance by employing just the associative, identity, and inverse properties of algebra. Associativity is conferred by flattening out immediate subexpressions with the same head. We could have given the attribute Flat to the functions, but then plus[x] would not reduce to x.

```
In[40]:= plus[times[u, x], plus[y, z]]
Out[40]= plus[times[u, x], plus[y, z]]

In[41]:= plus[a___, b_plus, c___] :=
             Flatten[Unevaluated[plus[a, b, c]], 1, plus]
         plus[a_] := a
         times[a___, b_times, c___] :=
             Flatten[Unevaluated[times[a, b, c]], 1, times]
         times[a_] := a

In[45]:= plus[times[u, x], plus[y, z]]
Out[45]= plus[times[u, x], y, z]
```

Very often we encounter expressions that contain subexpressions that reduce to zero or one (such as subtract[x, x]). It is probably useful to provide rules for these cases. The following evaluation rules use the identity and inverse properties of addition and multiplication:

```
In[46]:= plus[a___, 0, b___] := plus[a, b]
         plus[a___, 0., b___] := plus[a, b]
         subtract[a_, 0] := a
         subtract[a_, 0.] := a
         subtract[a_, a_] := 0
         times[a___, 1, b___] := times[a, b]
         times[a___, 1., b___] := times[a, b]
         divide[a_, 1] := a
         divide[a_, 1.] := a
         divide[a_, a_] := 1
```

Finally, here is the full effect:

```
In[56]:= plus[subtract[x, x], times[2, divide[plus[x, 2], 2]]]
Out[56]= times[2, divide[plus[x, 2], 2]]
```

Adam and Eve

The initial population in a GA is created by generating random character strings of a prescribed length, which is a fairly trivial task. In GP, on the other hand, we must generate random expression trees, which have not only a length (depth), but also breadth. The inputs to a random expression generator are the lists of functions and terminals, and the number of arguments that each function takes. (For simplicity, we restrict plus and times to two arguments even though *Mathematica* has no such restriction for Plus and Times.)

```
In[57]:= funcs = {{plus,2}, {subtract,2}, {times,2}, {divide,2}} ;
         terms := {x, y, z, Random[Integer, 3]} ;
```

We use recursion to construct an expression tree. The depth of the tree is controlled by decrementing a counter as we enter each level. The attribute HoldRest is necessary so that Random can be used in the terminal list.

```
In[59]:= randomElement[list_] :=
             list[[ Random[Integer, {1, Length[list]}] ]]

In[60]:= RandomExpression[depth_?Positive, funcs_, terms_] :=
             Module[{f, n},
                 {f, n} = randomElement[funcs] ;
                 If[n > 0,
                     f @@ Table[RandomExpression[depth - 1,
                         funcs, terms], {n}],
                   (* else *)
                     f
                 ]
             ]
         RandomExpression[1, funcs_, terms_] :=
             randomElement[terms]
         Attributes[RandomExpression] = {HoldRest} ;

In[63]:= RandomExpression[3, funcs, terms]
Out[63]= times[divide[x, y], plus[x, x]]
```

All the expressions produced in this manner have the same shape. They are full trees, that is, the distance from the root to each leaf is the same, and they always have a maximal number of nodes (before simplification). This is accomplished by restricting the use of terminals to the last level. Variably shaped trees are produced when terminals are allowed at any level. They can be generated using `RandomExpression` simply by including the terminals in the function set as functions of zero arguments.

```
In[64]:= comb = Join[funcs, {#, 0}& /@ terms] ;

In[65]:= re = RandomExpression[4, comb, terms]
Out[65]= times[plus[x, y], divide[times[y, x], plus[1, x]]]
```

The built-in function `Depth` and the recursive function `Size`, defined below, allow us to ascertain the structural diversity of the expressions in the constructed population. This is important because the early use of terminals may prevent even one branch of a tree from reaching the prescribed depth.

```
In[66]:= Size[_[args__]] := 1 + Plus @@ Map[Size, {args}]
         Size[_?AtomQ] := 1

In[68]:= Size[re]
Out[68]= 11
```

In constructing the initial population of expression trees, one should strive for variety. In the ramped-half-and-half method [Koza 1992, 93], the population is divided into equal groups for each depth, and half the expression trees in each group are full and the other half are not. The function `makePop` does this. The attribute `HoldAll` is necessary so that `Random` can be used in the terminal set.

```
In[69]:= makePop[funcs_, terms_, nPop_, maxDepth_, minDepth_:3] :=
           Module[{comb = Join[funcs, {#, 0}& /@ terms],
               d = maxDepth - minDepth + 1, i, depth, r},
             Table[depth = minDepth + Floor[(i-1)d/nPop] ;
               If[OddQ[i],
                 While[Depth[r =
                   RandomExpression[depth, funcs, terms]] < depth] ;
                 r,
                (* else *)
                 While[Depth[r =
                   RandomExpression[depth, comb, terms]] < depth] ;
                 r],
               {i, nPop}]
             ]
         Attributes[makePop] = {HoldAll} ;

In[71]:= pop = makePop[funcs, terms, 500, 7] ;
```

We can use `Depth` and `Size` to check that our generative method achieved the variety we sought. For a depth of seven and all the functions taking two arguments, there will be a maximum size of $2^7 - 1 = 127$ nodes.

```
In[72]:= Needs["Statistics`DataManipulation`"] ;

In[73]:= depths = Depth /@ pop ;
         d = Range[Min[depths], Max[depths]] ;
         c = CategoryCounts[depths, d] ;
         TableForm[Transpose[{d,c}],
             TableHeadings -> {None, {"depth", "# trees"}},
             TableSpacing -> {0,3}]
```

Out[76]//TableForm=

depth	# trees
3	100
4	100
5	100
6	100
7	100

```
In[77]:= sizes = Size /@ pop ;
         s = Range[0, 128, 8] ;
         c = RangeCounts[sizes, s] ;
         s = Transpose[{Drop[s, -1], Drop[s, 1]}] ;
         s = Apply[ToString[#1] <> " <=size< " <>
               ToString[#2]&, s, {1}] ;

In[82]:= TableForm[Transpose[{s, Take[c, {2, Length[c]-1}]}],
             TableHeadings -> {None, {"size", "# trees"}},
             TableSpacing -> {0, 3}]
```

Out[76]//TableForm=

size	# trees
0 <=size< 8	117
8 <=size< 16	155
16 <=size< 24	64
24 <=size< 32	54
32 <=size< 40	7
40 <=size< 48	8
48 <=size< 56	34
56 <=size< 64	11
64 <=size< 72	0
72 <=size< 80	0
80 <=size< 88	0
88 <=size< 96	3
96 <=size< 104	18
104 <=size< 112	26
112 <=size< 120	3
120 <=size< 128	0

Cain and Abel

The evolution of the population is carried out by applying the genetic operators to selected individuals to create offspring that become the next generation. The most frequently used genetic operators are reproduction and crossover; mutation is used less often. Reproduction, as the name suggests, is a direct copying of an individual expression from one generation to the next, crossover exchanges subexpressions from two parents to create two children, and mutation replaces a subexpression in an individual with a randomly generated subexpression. We also use a fourth operator, constant perturbation [Spencer 1994], in applications that employ constants as part of the terminal set.

Crossover

Crossover in GP is a bit more complicated than in a GA. Two parents are selected at random from the population. Then a subexpression is randomly selected from each parent and the two subexpressions are exchanged.

```
In[83]:= Adam = RandomExpression[3, funcs, terms]
Out[83]= plus[divide[z, x], times[y, 2]]

In[84]:= Eve = RandomExpression[4, funcs, terms]
Out[84]= divide[subtract[plus[1, 1], times[y, x]],
           divide[subtract[z, x], plus[z, 1]]]
```

Given two expression trees drawn on paper, a pair of scissors, and some glue, it is easy to perform a crossover. However, this is the trickiest function to program. The process can be divided into two parts: identifying the subexpressions to exchange, and swapping them. We use the built-in function `Position` to obtain a list of positions of all subexpressions, which are the parts that match the simple pattern `_` (`Blank`).

```
In[85]:= Position[Adam, _, Heads -> False]
Out[85]= {{1, 1,}, {1, 2}, {1}, {2, 1}, {2, 2}, {2}, {}}
```

We select at random one element of this list of parts and use the built-in function `Part` to extract the corresponding subexpression. `Part` takes a sequence of indices, so to use one of the lists returned by `Position`, the list has to be recast as a sequence.

```
In[86]:= Part[Adam, Sequence @@ %[[6]] ]
Out[86]= times[y, 2]
```

The built-in function `ReplacePart` can be used to insert the subexpressions. It takes as its third argument the position for replacement (we have to make a slight amendment to the function so that the whole expression can be replaced).

```
In[87]:= Unprotect[ReplacePart] ;
         ReplacePart[expr_, new_, {}] := new
         Protect[ReplacePart] ;
```

Here is the function that performs the crossover:

```
In[90]:= CrossOver[parents_] :=
           Module[{ind, sub},
             ind = randomElement[
               Position[#, _, Heads -> False]& /@ parents ;
             sub = MapThread[Part[#1, Sequence @@ #2]&,
                 {parents, ind}] ;
             MapThread[ReplacePart,
                 {parents, Reverse[sub], ind}] ]

In[91]:= {Cain, Abel} = CrossOver[{Adam, Eve}] ;

In[92]:= Cain
Out[92]= plus[divide[z, x], x]

In[92]:= Abel
Out[92]= divide[subtract[plus[1, 1], times[y, x]],
           divide[subtract[z, times[y, 2]], plus[z, 1]]]
```

As one can see in Figure 2, part `{2}` of `Adam` (`times[y, 2]`) was swapped with part `{2, 1, 2}` of `Eve` (`x`).

There are some interesting special cases that should be discussed. If the exchange takes place between two terminal nodes, the result is the same as for two mutations. Even if crossover takes place between two identical parents, the offspring will in general not be identical because the subexpressions selected for exchange will not necessarily be the same. This is in contrast to a GA where the fixed data structure forces the creation of two offspring not only identical to each other, but also identical to the parents.

During the evolution of the population, the depth of individuals tends to grow because subexpressions of different depths are exchanged. While this will allow great flexibility in finding a solution, the larger expressions take longer to evaluate. Therefore it is practical to impose an upper limit on the depth of an expression. In the event that a crossover will exceed this limit, the offending offspring is replaced with one of its parents [Koza 1992, 104], as shown in this revised definition of `CrossOver`:

```
In[93]:= Clear[CrossOver] ;
         CrossOver[parents_, maxDepth_:17] :=
           Module[{ind, sub, children},
             ind = randomElement[
                  Position[#, _, Heads -> False]]& /@ parents ;
             sub = MapThread[Part[#1, Sequence @@ #2]&,
                 {parents, ind}] ;
             children = MapThread[ReplacePart,
                 {parents, Reverse[sub], ind}] ;
             MapThread[
                 If[Depth[#1] <= maxDepth, #1, #2]&,
                 {children, parents}] ]
```

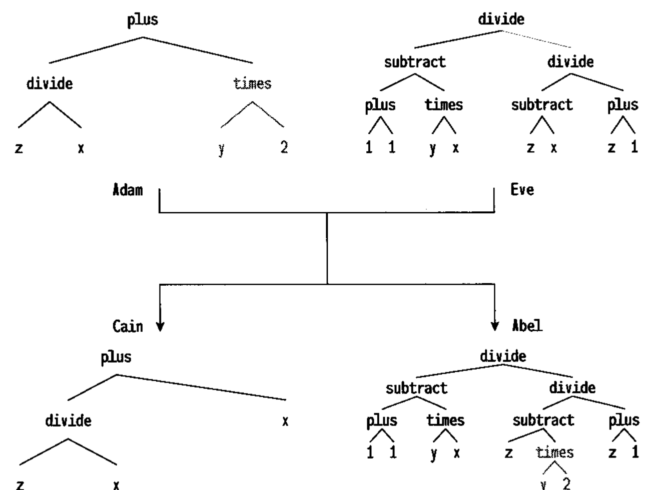


FIGURE 2. The crossover of Adam and Eve to produce Cain and Abel.

Mutation

In the traditional GA, mutation merely replaces one randomly selected gene with another one from the allowed alphabet. Typically, a bit is flipped, as most GAs use a binary representation. In genetic programming, mutation replaces a randomly selected subexpression with a new, randomly generated subexpression. The depth of the new subexpression can be anywhere from zero (a terminal) to some maximum based on the overall maximum depth for an expression, which may be larger than the orignal subexpression.

```
In[95]:= Mutate[expr_, funcs_, terms_, maxDepth_:17] :=
           Module[{pos, ind, depth, newSub},
             ind = randomElement[
                  Position[expr, _, Heads -> False]] ;
             depth = Random[Integer,
                 (maxDepth - 1 - Length[ind])/4] + 1 ;
             newSub = RandomExpression[depth, funcs, terms] ;
             ReplacePart[expr, newSub, ind] ]
         Attributes[Mutate] = {HoldRest} ;

In[97]:= Simon = RandomExpression[3, funcs, terms]
Out[97]= times[divide[y, z], subtract[1, z]]

In[98]:= Peter = Mutate[Simon, funcs, terms]
Out[98]= times[divide[y, z], subtract[times[z, y], z]]
```

In Figure 3, we see that part {2,1} of Simon, of depth 1, was replaced by a subexpression of depth 2 to become Peter.

Constant Perturbation

In many instances, an expression tree can be improved by making a small change in one of the component constants. Fine tuning of this sort is often more efficient than relying on mutation to achieve the same effect. We use the pattern _?NumberQ to identify the terminal parts of an expression tree that hold a constant numeric value.

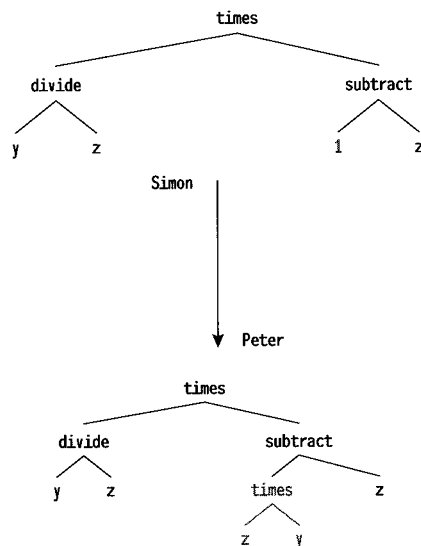


FIGURE 3. The mutation of Simon to produce Peter.

```
In[99]:= Saul = RandomExpression[4, funcs,
                 {x, Random[Integer, {-5, 5}]}]
Out[99]= divide[times[subtract[4, x], 3, -1],
           subtract[subtract[-5, x], subtract[3, x]]]
```

Note that we have used a larger set of terminals in this example. Here are the positions of the numeric terminals.

```
In[100]:= Position[Saul, _?NumberQ]
Out[100]= {{1, 1, 1}, {1, 2}, {1, 3}, {2, 1, 1}, {2, 2, 1}}
```

Here are their values.

```
In[101]:= Part[Saul, Sequence @@ #]& /@ %
Out[101]= {4, 3, -1, -5, 3}
```

Now, one of these nodes is randomly selected and its value is perturbed by a small amount. Note that Real, Integer, Rational, and Complex constants are replaced by values of the same type.

```
In[102]:= perturbConstant[n_Real] := Random[Real, n {0.8, 1.25}]
          perturbConstant[n_Integer] := Random[Integer,
              {Min[n-3, Floor[0.80 n]],
                  Max[Ceiling[1.25 n], n+3]}] /; n >= 0
          perturbConstant[n_Integer] := Random[Integer,
              {Min[n-3, Floor[1.25 n]],
                  Max[Ceiling[0.80 n], n+3]}] /; n < 0

In[105]:= Perturb[expr_]:=
            Module[{constants, ind},
              constants = Position[expr, _?NumberQ] ;
              If[Length[constants] > 0,
                  ind = randomElement[constants] ;
                  ReplacePart[expr,
                      perturbConstant[expr[[Sequence @@ ind]]],
                      ind],
                (* else *)
                  expr
              ] ]

In[106]:= Paul = Perturb[Saul]
Out[106]= divide[times[subtract[1, x], 3, -1],
            subtract[subtract[-5, x], subtract[3, x]]]
```

Figure 4 shows that the value 4 at part {1, 1, 1} of Saul was replaced with the value 1 in Paul.

Fitness and Selection

How does one judge the "fitness" of an individual expression tree? We need to measure how well the program represented by the expression solves the problem at hand. The better the program performs, the more fit it is and thus the more likely it is to survive to the next generation or mate with other individuals. The progression from performance to probability of selection is controlled by a well-defined series of fitness calculations [Koza 1992, 94–98].

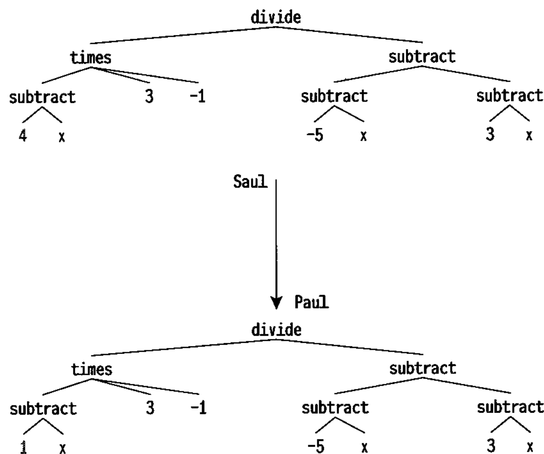


FIGURE 4. The perturbation of Saul to produce Paul.

For the problem of generating a particular sequence, we have a set of observations or cases against which we can measure the performance of the program.

```
In[107]:= seq = {5, 31, 121, 341, 781, 1555, 2801, 4681};
```

First, we couple the sequence with the values of the independent variable *i* to make a list of pairs. (For problems with more than one independent variable, the data is usually obtained as an array.)

```
In[108]:= cases = Transpose[{Range[8], seq}]
Out[108]= {{1, 5}, {2, 31}, {3, 121}, {4, 341}, {5, 781},
          {6, 1555}, {7, 2801}, {8, 4681}}
```

Next, for computational efficiency, we separate the observations into the independent and dependent parts.

```
In[109]:= prepData[cases_, vars_] :=
            Transpose[{Drop[#,-1], Last[#]}& /@ cases] /;
            Length[First[cases]] == Length[vars] + 1
          prepData[cases, {i}]
Out[109]= {{{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}},
          {5, 31, 121, 341, 781, 1555, 2801, 4681}}
```

In this case, the performance of a program is measured by an error function, which compares the given data values with the values computed by the program. We could use the sum of absolute errors or the sum of squared errors over the fitness cases, and there is no compelling reason to choose one over the other. The functions SumAbs and SumSqr compute the sums of absolute error and squared error for a list of expressions.

```
In[110]:= SumAbs[{x_, y_}, vars_, exprs_] :=
           Module[{f},
             (f = Function @@ {vars, Eval[#]} ;
             Plus @@ Abs[y - Apply[f, x, {1}]] // N)& /@ exprs ]
```

```
In[111]:= SumSqr[{x_, y_}, vars_, exprs_] :=
           Module[{f, d},
             (f = Function @@ {vars, Eval[#]} ;
             d = y - Apply[f, x, {1}] // N ;
             d.d)& /@ exprs ]
```

To demonstrate how we select individual expressions using a measure of the error, we'll first generate a small population using the following sets of functions, terminals, and variables.

```
In[112]:= funcs = {{plus,2}, {subtract,2}, {times,2}, {divide,2}} ;
          terms := {i, Random[Integer, {-3, 3}]} ;
          vars = {i} ;
```

```
In[115]:= data = prepData[cases, vars] ;
```

```
In[116]:= pop = makePop[funcs, terms, 10, 6] ;
```

The list of errors obtained by applying SumAbs to the population is known as the *raw* fitness.

```
In[117]:= rawFit = SumAbs[data, vars, pop]
Out[117]= {1.47574 10^20, 10280., 7.3787 10^19, 10208., 10316.,
          11420., 10298.6, 2.78531 10^18, 1.68461 10^6, 7.3787 10^19}
```

It is always possible to minimize the error by adding more adjustable parameters to the fitting function, and we have done nothing to protect against this so-called overfitting. One could build parsimony into the fitness function by inflating the error proportional to the number of terminals or functions in the expression tree. However, it has been shown that penalizing the fitness by the expression size seriously degrades performance [Koza 1992, 612–614]. Therefore, it is more convenient to enforce parsimony via the genetic operators (maximum depth) and algebraic simplification.

Depending on the problem, we may want either to minimize or maximize the raw fitness. To simplify things, the raw fitness is converted to *standardized* fitness, a non-negative quantity which is always minimized. When raw fitness is error, we can let the standardized fitness be the same as the raw fitness.

```
In[118]:= standardizedFit[rawfit_] := rawfit
```

```
In[119]:= standFit = standardizedFit[rawFit] ;
```

When the raw fitness should be maximized (such as a score in a game), the following definitions may be appropriate. If there is no theoretical upper bound to the score, maxScore may be set to the maximum value of the population's raw fitness, or to some other practical value.

```
standardizedFit[rawfit_] := maxScore - rawfit
standardizedFit[rawfit_] := Max[rawfit] - rawfit
```

The standardized fitness is inverted to give the *adjusted* fitness, which is proportional to the probability of selection.

```
In[120]:= adjFit = 1 / (1 + standFit)

                 -21                         -20
Out[120]= {6.77626 10   , 0.0000972668, 1.35525 10   , 0.0000979528,

                                                   -19
  0.0000969274, 0.000087558, 0.0000970908, 3.59026 10   ,

          -7           -20
  5.93609 10  , 1.35525 10   }
```

There are three main types of selection. In *fitness-proportionate* selection (or *roulette wheel* selection), the probability of selection is directly proportional to the magnitude of the fitness. This type of selection is most easily accomplished by using a list of cumulative sums of adjusted fitnesses [Freeman 1993].

```
In[121]:= cumSum = FoldList[Plus, 0, adjFit] // Rest

                 -21
Out[121]= {6.77626 10   , 0.0000972668, 0.0000972668, 0.00019522,
  0.000292147, 0.000379705, 0.000476796, 0.000476796,
  0.000477389, 0.000477389}
```

We generate a random number between zero and the final cumulative sum, and find the position of the first cumulant that exceeds it. The function selectOne returns the index of the selected individual.

```
In[122]:= selectOne[cumProb_] :=
    Module[{r = Random[] Last[cumProb]},
      Position[cumProb, _?(# >= r &), {1}, 1][[1, 1]] ]

In[123]:= selectOne[cumSum]

Out[123]= 4

In[124]:= pop[[%]]

Out[124]= divide[times[-1, i], divide[subtract[0, -1], -3]]
```

This functional definition of selectOne is not very fast because each element of the list of cumulative fitnesses must be examined in turn until one matches the pattern. The algorithm is of order $O(n)$. Because the list is nondecreasing, we can use a more efficient binary search algorithm of order $O(\log_2 n)$. The following procedural binary search version is more than three times faster for a list of 500 elements.

```
In[125]:= selectOne[cumProb_] :=
    Module[{lo = 1, hi = Length[cumProb], mid,
            r = Random[] Last[cumProb]},
      While[lo != hi-1,
        mid = Round[(lo + hi)/2] ;
        If[r < cumProb[[mid]], hi = mid, lo = mid]
      ] ;
      If[r <= cumProb[[lo]], lo, hi] ]
```

Rank selection is closely related to fitness proportionate selection. In this method, the probability of selection is proportional to only the rank, or order, of the adjusted fitness, and not its actual magnitude. Rank selection is used when one wants to enhance the distinction between individuals of nearly equal fitness. The function toRank takes a list of adjusted fitnesses and returns their ranks (where 1 is low and ties are allowed).

```
In[126]:= toRank[adjFit_] :=
    Module[{f = Union[adjFit, SameTest -> SameQ], c,
            r = Range[Length[adjFit]], g, b, nr},
      c = CategoryCounts[adjFit, f] ;
      g = (b = Take[r, #]; r = Drop[r, #]; b)& /@ c ;
      nr = (Plus @@ #)/Length[#]& /@ g ;
      nr[[Flatten[Position[f, #]& /@ adjFit]]] ]

In[127]:= toRank[adjFit] // N

Out[127]= {1., 9., 2.5, 10., 7., 6., 8., 4., 5., 2.5}
```

The cumulative sum of ranks is used in place of the cumulative sum of adjusted fitnesses in selectOne.

```
In[128]:= cumSum = FoldList[Plus, 0, %] // Rest

Out[128]= {1., 10., 12.5, 22.5, 29.5, 35.5, 43.5, 47.5, 52.5, 55.}
```

The third method of selection is *tournament* selection. In this method, a set of individuals is drawn at random from the population and the most fit individual is selected.

Finally, we need to define a success predicate that returns True when an optimal or, depending on the problem, near-optimal solution is found. When the raw fitness is error, a perfect solution has zero error; otherwise, we might accept an error below a specified threshold, as in this example:

```
successQ[rawfit_] := Or @@ Thread[rawfit < 0.5]
```

GeneticProgram

The function GeneticProgram combines the pieces of code we have developed (an excerpt is shown in Listing 1). It takes as arguments the lists of cases, variables, functions, and terminals; the function to preprocess the fitness cases; the standardized fitness function; and the success predicate function. The numerical values for the population size and the maximum number of generations over which the population is allowed to evolve, as well as the list of genetic operators, can be changed from their default values with the use of options. Our function generates an initial population and evolves successive generations. The result returned by GeneticProgram is a list of replacement rules which includes the following expressions: BestOfRun is a couplet of the raw fitness and the expression for the best individual found during the entire course of the run, FinalFit is a list of the standardized fitnesses for the final population, and FinalPop is the corresponding list of expressions. In many of the problems we encounter in medicinal chemistry, there is no one correct answer, so we prefer to look at an ensemble of good answers.

We have also included Print statements so that the evolution of the population can be monitored. The generation (g), best over-all standardized fitness (run best), cpu time used (cpu), and the memory used (mem) are reported periodically.

```
data = prepData[cases, varlist] ;

stdFit = standardFit[data, varlist, pop] ;
adjFit = 1 / (1 + stdFit) ;

best = Position[adjFit, Max[adjFit]][[1, 1]] ;
bestOfRun = {adjFit[[best]], pop[[best]]} ;

While[g < maxGen && !successQ[stdFit],
    probExpr = FoldList[Plus, 0, adjFit] // Rest,
    newPop = Table[Null, {popSize}] ;
    i = 0 ;
    While[i < popSize,
        {op, np} = genOpers[[ selectOne[probOper] ]] ;
        parents = pop[[ Table[selectOne[probExpr], {np}] ]] ;
        children = op[parents] ;
        For[j = 1, j <= np && i < popSize, j++,
            newPop[[++i]] = children[[j]] ;
            ] ;
        ] ;

    pop = newPop ;
    g++ ;
    stdFit = standardFit[data, varlist, pop] ;
    adjFit = 1 / (1 + stdFit) ;

    best = Position[adjFit, Max[adjFit]][[1, 1]] ;
    If[adjFit[[best]] > First[bestOfRun],
        bestOfRun = {adjFit[[best]], pop[[best]]}] ;
    ] ;
```

LISTING 1. Part of the definition of GeneticProgram.

The genetic operators are specified in a list, together with their numbers of input expressions and their relative frequencies. This format allows us to draw any one of them at random and apply it to the appropriate number of individuals drawn from the population. The relative frequencies do not indicate the frequency with which the operators are invoked, but rather the frequency with which new individuals are created. Therefore, to obtain the necessary probability of using an operator, the relative frequency is divided by the number of parents (which is the same as the number of offspring produced).

The asexual genetic operators return a single expression, while the sexual one (crossover) returns a list of expressions. One of the great strengths of *Mathematica* is its ability to be customized on the fly. Thus we are able to recast Mutate and Perturb to return lists of one element, and define Reproduce with the following snippet of code:

```
opers = opers /.
  {Reproduce -> Identity,
   Mutate -> ({Mutate[Sequence @@ #, funcs, terms]}&),
   Perturb -> ({Perturb[Sequence @@ #]}&)} ;
```

As each generation is completed, the standardized fitness (stdFit) is computed and converted to the adjusted fitness (adjFit). The heart of GeneticProgram is in the two nested While loops, the outer one over generations and the inner one over individual expressions. In the outer loop, the cummulative sum of adjusted fitnesses (probExpr) is computed and an empty next generation (newPop) is made. Then, in the inner loop, a genetic operator (op) is selected at random, the parents are selected, and finally the children are created and inserted into the new generation. All these steps are done under the control of the two lists of cummulative probabilities, probOper for the genetic operators and probExpr for the parents. At the end of the outer loop, pop is replaced by newPop and new fitnesses are computed.

Symbolic Regression

Let us return to our problem of finding an expression to generate a given sequence. We first start a new *Mathematica* kernel and load our packages.

```
In[1]:= Needs["GeneticProgramming`"]
        GeneticProgramming.m, version 2.1.

In[2]:= Needs["SymbolicRegression`"]
        SymbolicRegression.m, version 1.0.
```

The data for the problem is a list of eight observations:

```
In[3]:= seq = {5, 31, 121, 341, 781, 1555, 2801, 4681};
```

We choose a function set containing just the four arithmetic functions (nothing in the data warrants the use of trigonometric functions or logarithms). The independent variable i and small integers comprise the terminal set.

```
In[4]:= funcs = {{plus,2}, {subtract,2}, {times,2}, {divide,2}} ;
        terms := {i, Random[Integer, {-3, 3}]}
```

As standardized fitness is the same as raw fitness, we will use SumSqr to calculate it. The success predicate will be a perfect fit.

```
In[6]:= successQ[rawfit_] := Or @@ Thread[Chop[rawfit] == 0]
```

We will use the default population size of 500, a typical value. This number may seem rather high for such a simple problem, but it is required to allow the population to maintain the necessary diversity. It is difficult, however, to predict how the population size scales with the complexity of the problem. Suffice it to say here that we have solved a three-variable problem with the same population size.

```
In[7]:= result = GeneticProgram[seq, {i}, funcs, terms, GetData,
            SumSqr, successQ, GeneticOperators ->
              {{CrossOver, 2, .85}, {Reproduce, 1, .01},
               {Mutate, 1, .01}, {Perturb, 1, .13}}] ;

Out[7]= g=0, run best = 480416., cpu = 0:02:26, mem = 2877 K
        g=5, run best = 12056., cpu = 0:08:01, mem = 2824 K
        g=10, run best = 12056., cpu = 0:12:41, mem = 2812 K
        g=15, run best = 6056., cpu = 0:17:28, mem = 2816 K
        g=20, run best = 8., cpu = 0:22:48, mem = 2886 K
        g=25, run best = 4.09171, cpu = 0:29:33, mem = 2900 K
        g=27, run best = 0, cpu = 0:32:22, mem = 2928 K
```

Here is the best-fitting expression and its depth:

```
In[8]:=  b = BestOfRun /. result
Out[8]= {0, times[divide[i, divide[i,
           divide[i, divide[i, times[i, i]]]]],
         subtract[subtract[plus[i, -3, i,
            subtract[subtract[plus[2, 2],
              divide[times[-1, subtract[i,
                 divide[times[-1, i], i]]],
               times[i, i]]], times[0, -1, i]]], times[-1, i]],
          times[subtract[2, i], i]]]}

In[9]:=  Depth[Last[b]]
Out[9]= 12
```

Here is the simplified form of the expression:

```
In[10]:= Eval[Last[b]]

                                                 2
Out[10]= PDivide[i, PDivide[i, PDivide[i, PDivide[i, i ]]]]

                                                           2
           (1 + 3 i - (2 - i) i - PDivide[-i + PDivide[-i, i], i ])

In[11]:= bestFit = % /. PDivide -> Divide // Expand // Together

                 2    3    4
Out[11]= 1 + i + i  + i  + i
```

Now, let's look at the runners up. Sometimes the numerically best solution does not make good physical, chemical, or biological sense for the problem, so it is always a good idea to examine plausible alternatives. In the real world, where experimental error lurks, these second- and third-best answers may well be better than the best-fit solution when one also takes into account parsimony, the physical interpretation of the resulting expressions, and test cases that were not used in training.

```
In[12]:= fp = Sort[Transpose[{FinalFit, FinalPop} /. result]] ;
```

Here are the fitnesses of the five best expressions.

```
In[13]:= First /@ Take[fp, 5]
Out[13]= {0, 4.09171, 6.02694, 8., 8.}
```

Here is the second-best expression and its depth:

```
In[14]:= sb = fp[[2]]
Out[14]= {4.09171, times[divide[i, divide[i,
            divide[i, divide[i, times[i, i]]]]],
          subtract[subtract[plus[i, -3, i,
             subtract[subtract[plus[2, 2],
               divide[times[-1, subtract[i,
                  divide[times[-1, i], times[i, i]]]],
                  times[i, i]]],
             times[0, -1, i]]], times[-1, i]],
                  times[subtract[2, i], i]]]}
```

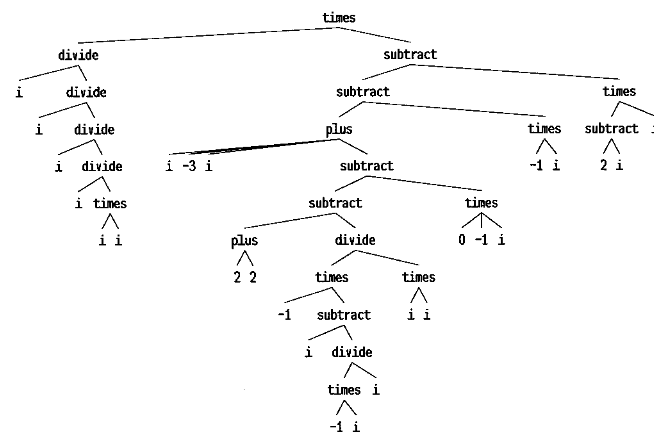


FIGURE 5. The best-fitting expression found by GeneticProgram.

```
In[15]:= Depth[Last[sb]]
Out[15]= 12
```

Here is the simplified form of this expression, which differs from the best expression by the additional term $(1 - i)/i$.

```
In[16]:= Eval[Last[sb]]

                                                 2
Out[16]= PDivide[i, PDivide[i, PDivide[i, PDivide[i, i ]]]]

                                                           2    2
           (1 + 3 i - (2 - i) i - PDivide[-i + PDivide[-i, i ], i ])

In[17]:= secondBestFit = % /. PDivide->Divide // Expand //
            Together

                2    3    4    5
          1 + i  + i  + i  + i
Out[17]= ---------------------
                   i

In[18]:= secondBestFit - bestFit // Simplify

               1
Out[18]= -1 + -
               i
```

Suppose now that we are given the next four entries in the sequence to test our results: 7381, 11111, 16105, and 22621. (If one cannot obtain new data with which to validate the results, one should hold back a portion of the data from the fitting process for this purpose.) Let's compare these entries with the numbers generated by the best-fit expression.

```
In[19]:= (bestFit /. i -> #)& /@ Range[9, 12]
Out[19]= {7381, 11111, 16105, 22621}
```

Viola! Still a perfect fit. Here is a comparison of the two best results:

```
In[20]:= TableForm[
           {#, (bestFit /. i -> #),
             N[secondBestFit /. i -> #]}& /@ Range[12],
           TableHeadings -> {None, {"i", "best", "second best"}},
           TableSpacing -> {0, 3}]
```

 © 1995 Miller Freeman Publications

Out[20]//TableForm=

i	best	second best
1	5	5.
2	31	30.5
3	121	120.333
4	341	340.25
5	781	780.2
6	1555	1554.17
7	2801	2800.14
8	4681	4680.12
9	7381	7380.11
10	11111	11110.1
11	16105	16104.1
12	22621	22620.1

Closing Remarks

The functions described here, as well as several others that should be useful, are collected in the packages `GeneticProgramming.m` and `SymbolicRegression.m`. The options and output have been expanded to include information we have found helpful in deciding whether or not to continue a run or to start over with a new population. We have attempted to isolate the problem-specific functions and definitions in `SymbolicRegression.m` and to keep `GeneticProgramming.m` as generic as possible.

Currently, we exploring ways to evolve parsimonious expressions, to employ smaller but more diverse populations, and to incorporate analytic methods for determining the values of constants.

Genetic programming is field of active research, and there are many issues and techniques we have not touched on here. The reader is referred to [Koza 1992; 1994] for further reading and references. The Internet news group `comp.ai.genetic` and the mailing list `genetic-programming@cs.stanford.edu` are devoted to evolutionary computing. These World Wide Web sites may also be of interest: `http://www-mitpress.mit.edu/jrnls-catalog/evolution.html` and `http://gal4.ge.uiuc.edu/illigal.home.html`.

Acknowledgments

The author wishes to acknowledge many helpful suggestions from the editor, and advice from Alan DeGuzman and Robby Villegas at the technical support department at Wolfram Research, Inc. He also wishes to thank Mary Curran Nachbar, a high school student, for careful reading of the manuscript to identify vague and imprecise terms and phrases.

References

Davis, L. 1991. *Handbook of Genetic Algorithms.* New York: Van Nostrand Reinhold.

Freeman, J. 1993. Simulating a Basic Genetic Algorithm. *The Mathematica Journal* 3(2): 52–56.

Gaylord, R.J., S.N. Kamin, and P.R. Wellin. 1993. *Introduction to Programming with Mathematica.* TELOS/Springer-Verlag.

Goldberg, D.E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.

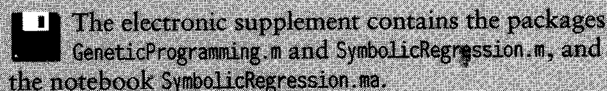
Koza, J.R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

Koza, J.R. 1994.*Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press.

Lee, G.Y. 1994. Private communication.

Spencer, G. 1994. Automatic Generation of Programs for Crawling and Walking. In *Advances in Genetic Programming,* ed. Kenneth E. Kinnear, Jr. MIT Press.

Robert B. Nachbar
Molecular Design and Diversity Department,
Merck Research Laboratories,
P.O. Box 2000, Rahway, NJ 07065
nachbar@merck.com


The electronic supplement contains the packages `GeneticProgramming.m` and `SymbolicRegression.m`, and the notebook `SymbolicRegression.ma`.