A C++ Implementation of Genetic Programming

Article · January 1995

Source: CiteSeer

1 author:

Larry Gritz
Sony Pictures Imageworks
21 PUBLICATIONS **280** CITATIONS

SEE PROFILE

A C++ Implementation of Genetic Programming

Larry I. Gritz

Department of EE & CS
The George Washington University
Washington, DC 20052

Email: gritz@seas.gwu.edu

1 July, 1993

ABSTRACT

Genetic Programming (GP) is an evolutionary approach to optimization. GP is similar to Genetic Algorithms (GA), but operates on computer programs rather than on fixed length binary strings. GP is commonly implemented using the LISP programming language. We felt that there was a need for the GP algorithm to be available as a toolbox routine, callable from programming languages other than LISP.

This document presents a very abbreviated introduction to the GP paradigm. The use of the libgp library, callable from any C++ program, is documented. The implementation issues surrounding our C++ implementation of the GP paradigm are discussed. Several working example C++ programs using this library are presented with documentation and discussion.

Table of Contents

1. INTRODUCTION

1.1. Genetic Algorithms

Genetic Algorithms (GA's) are a method of optimization which uses an evolutionary metaphor [8][7]. The GA approach to optimization is typically to encode potential solutions to the problem as fixed length binary strings. A *population* of these *individuals* are evaluated by a *fitness function* (an objective function which measures how well a potential solution will solve the problem).

Members of the population may be selected to be included in the next *generation*. Individuals are selected stochastically, with their chance of selection being proportional to their fitness value. The result of this *reproduction* operation is for the subsequent generation to be populated with the strings with higher average fitness values, and for the strings with low fitness values to die off---survival of the fittest.

New strings may be made using the genetic *crossover* operation, which mixes the genetic material of two individuals. This sometimes results in strings which are even more fit (i.e. solve the stated problem better) than their parents. In addition, an asexual *mutation* operator which purposely introduces errors into the genetic material of an individual (by flipping a bit) may occasionally produce a mutated individual which is more fit than its progenitor.

The GA approach is surprisingly robust and efficient at finding global minima to certain optimization problems. The power of GA is especially clear with problems which are highly nonlinear, or have complicated coupled interactions of parameters.

1.2. Genetic Programming

Genetic Programming (GP) is an evolutionary metaphor similar to GA. The major difference is that GA operates on fixed length binary strings, while GP operates on computer programs of varying complexity. One advantage to this approach is that often, the complexity of an optimal solution to a problem is not known *a priori*. With GA, the complexity (length of the binary strings) must be specified at the start. GP, on the other hand, can evolve a complexity level appropriate to the problem.

A surprising number of optimization problems can be stated as problems of *program induction*. That is, we know that the solution to the problem is an appropriate formula or computer program, but we are not sure how to write the program.

GP is commonly implemented using the LISP programming language. LISP is convenient because programs, data structures, and GP individuals (represented as parse trees), are all identical data structures in LISP. However, we felt that there was a need for the GP algorithm to be available as a toolbox routine, callable from programming languages other than LISP.

This document presents a very abbreviated introduction to the GP paradigm. The use of the `libgp` library, callable from any C++ program, is documented. The implementation issues surrounding our C++ implementation of the GP paradigm are discussed. Several working example C++ programs using this library are presented with documentation and discussion.

1.3. About This Document

In Chapter 2 we present a very brief introduction to Genetic Programming. The basic GP operations are presented so that the casual reader can follow the rest of this document.

A user manual for `libgp`, our C++ implementation of the Genetic Programming Paradigm, is given in Chapter 3. The C++ implementations of two GP problems from [9] are listed.

Source code and explanation for the `libgp` library may be found in Chapter 4. This is not necessarily a complete library, but should serve as a starting point for those who wish to study GP using the C++ language.

Chapter 5 lists the source code for a sample application of Genetic Programming--simple symbolic regression. From this program listing, it should be apparent how Genetic Programming may be used as a tool which can be called from another application.

Finally, a short bibliography on material related to GA and GP can be found at the end of this document. While I make no attempt to provide a comprehensive bibliography, the citations may provide a good place to start learning about GA and GP. This author highly recommends [7] as a starting point for GA, and [9] as the only comprehensive work on GP. Both of these works contain comprehensive bibliographies of GA- and GP-related literature.

2. A GP Primer

Below, I will present an extremely brief explanation of the basics of the GP paradigm, the necessary elements of a GP specification, and the GP operations. I will not touch on any analysis, justification, or theory of GP. Koza's book [9] is a comprehensive treatment of the GP paradigm, and should be referred to for any further in-depth description and analysis of GP. The remainder of this report will use terminology and examples from [9] with minimal explanation.

2.1. The Basic GP Algorithm

The basic GP algorithm is simple, and may be explained very briefly. We have a particular optimization problem, the solution of which is a computer program. Given an arbitrary computer program, we need a way to evaluate how well that program solves our problem. This is called a *fitness measure*.

We start with a *population* of randomly generated programs, and evaluate their fitness values. We should expect that the fitness values of these random individuals will be quite low, since the chances are slim that a random computer program will solve our problem exactly. This initial population is the first generation.

Next, we form the population for the second generation in several ways:

- Reproduction, which copies (without alteration) individuals from the previous generation that had high fitness values.
- Crossover, which combines two individuals from the previous generation to yield two new individuals.
- Mutation, permutation, and encapsulation, which copy individuals with high fitness from the previous generation, making small changes during the copy.

Since the individuals involved in these operations are typically selected in some type of fitness-proportionate manner, unfit individuals will tend not to make it to the next generation while highly fit individuals will tend to be present in the next generation. The alterations resulting from crossover, mutation, permutation, and encapsulation will sometimes result in individual programs which are more fit than any programs seen in previous generations.

This cycle is repeated for each generation until a maximum number of generations is reached, or an individual is found which adequately solves the problem at hand. What is found is that each generation has individuals with higher

fitness than the previous generation, and that this often results in finding optimal individuals which could not have been found by mere blind random search.

The highest level of the GP algorithm can be formalized as below:

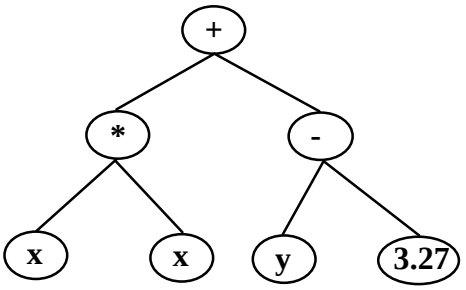
```
let M = the population size;
let G = the maximum number of generations to run;
create M random programs for generation 1;
for gen = 1 to G do:
   evaluate the fitness of all the individuals;
   if one solves the problem adequately, then
      terminate;
   construct the next generation of individuals;
 endfor;
report the best individual found.
```

Since the Genetic Programming paradigm is driven by stochastic processes such as the initial population of random programs and the probabilistic selection of individuals for reproduction and crossover, it is not guaranteed that a solution to our problem will be solved in any particular run. However, we can perform multiple independent runs (with different random number seeds on each run) until we find a solution that is adequate.

2.2. Individuals

What is the nature of the individual programs upon which we are operating? Though we could operate on any programming language representation, for simplicity we operate on program *parse trees*. Textually, these parse trees can be represented by LISP S-expressions [11]. Figure 1 shows a typical parse tree and its associated S-expression. The parse tree and S-expression correspond to the following formula in algebraic notation:

$$x^2 + (y - 3.27)$$



(+ (* x x) (- y 3.27))

Figure 1: A parse tree and its corresponding S-expression

Notice that there are two types of entities in the parse tree. Leaf nodes are either constants, named variables, or functions which take no arguments. These are called *terminals*. Non-leaf nodes are always functions which take one or more arguments.

Genetic Programming is typically implemented using LISP. One reason for this is that in LISP, programs, their parse trees, and their textual representations are all represented by S-expressions. Fundamentally, of course, there is nothing special about LISP that makes it necessary for GP.

We have chosen to implement our system in C++, and to present it as a library which can be called from within other C++ programs. However, the genetically evolved programs are still S-expressions, not C++ programs. This is because S-expressions still provide a compact representation which is easy to parse and whose parse trees are easy to manipulate.

2.3. The Terminal and Function Sets

The parse trees which we manipulate are assembled from a pre-chosen set of terminals and functions. The parse tree of Figure 1 utilized the terminals "x" and "y", and the floating point constant 3.27. Also, the functions *, +, and - were used, corresponding to the multiplication, addition, and subtraction operators, respectively.

One aspect of running the GP algorithm is determining the sets of terminals and functions which are needed to solve the problem. We do not want terminals and functions which are not needed to solve the problem, since they will only slow down our search for the correct solution. On the other hand, if a particular problem's unknown solution requires a function which is not in our set, we will never find the correct solution. We need to make our best guess as to which terminals and functions to use, perhaps erring slightly on the side of too many functions rather than too few.

2.4. Building the Initial Population

When creating random individuals for the initial generation, we must create a parse tree from our terminal and function sets. We can start by randomly choosing a function from our function set to serve as the root node. Each function has a fixed number of arguments which it requires. The addition operator, for example, requires two arguments.

For each argument required by the function we have chosen for the root node, we choose a node randomly from the union of the terminal and function sets. If it is a terminal which is chosen, that branch of the tree ends there. If it is another

function, we choose its arguments recursively, until the entire tree is filled in. We typically want a maximum depth of the initial trees, so at the maximum level we may restrict our choices to the terminal set.

The method of creating trees described above is called the "grow" method by Koza. Koza also describes the "full" method of creating trees. In the "full" method, trees are all full trees. That is, all nodes at the maximum depth are terminals and all nodes at depths less than the maximum are functions.

Koza also describes another method which is favored over both the "grow" and "full" method, which he calls the "ramped half-and-half" method. In this method, an equal number of trees is created with depths ranging between 2 and the maximum allowable depth. For example, if the maximum depth is 6, then there will be an equal number of trees created with depths 2, 3, 4, 5, and 6. For each depth level, 50% of the trees are created using the "grow" method and 50% are created using the "full" method. Koza argues that this will result in the initial population having the widest variety of tree shapes and sizes and will facilitate a rapid solution.

For some problems, the solution may require various numerical constants. Since the terminal set consists of variable names, these constants are not present. Koza proposes using a special terminal called the *ephemeral random constant*. Each time this terminal is selected in the creation of the initial population, it is replaced with a random constant. The type of constant (i.e. real or integer) and its range may be problem-dependent.

2.5. Fitness Measures

When all of the initial individuals are created, they must have their fitnesses rated. The problem-dependent fitness evaluation function returns a fitness value called the *raw fitness* of that individual. *Standardized fitness* is a measure of fitness in which more fit individuals have lower standardized fitness values than less fit individuals. Ideally, an optimal individual should have a standardized fitness value of 0. For some cases, raw fitness and standardized fitness may be identical. For other cases, there may be a necessary transformation between raw fitness and standardized fitness in order to get the standardized fitness values to meet this specification.

Adjusted fitness is a measure given by the following formula:

$$f_{adj} = \frac{1}{1 + f_{std}}$$

Finally, *normalized fitness* is the adjusted fitness value divided by the total of the adjusted fitnesses of the entire population:

$$f_{nor} = \frac{f_{adj}}{\sum_{k=1}^{M} f_{adj\,k}}$$

2.6. Creating Subsequent Generations

Once a population has been created and the fitness of all its individuals has been evaluated, it is time to create the next generation. We must generate *M* new individuals, where *M* is the number of individuals in the population. We primarily generate new individuals by reproduction and crossover. In addition, we may sometimes use mutation, permutation, editing, and encapsulation, though Koza reports that use of these operations does not tend to increase overall performance of the GP algorithm.

Reproduction involves selecting a single individual from the old population and copying it without alteration into the new population. The selection of the individual is fitness-proportionate, which means that the chance of a particular individual being selected is proportional to its normalized fitness.

Crossover selects two individuals from the old population, typically using fitness-proportionate selection. These two individuals are combined by randomly selecting a node from each parse tree, then swapping the subtrees below that node. Figure 2 illustrates two parse trees before and after the crossover operation.

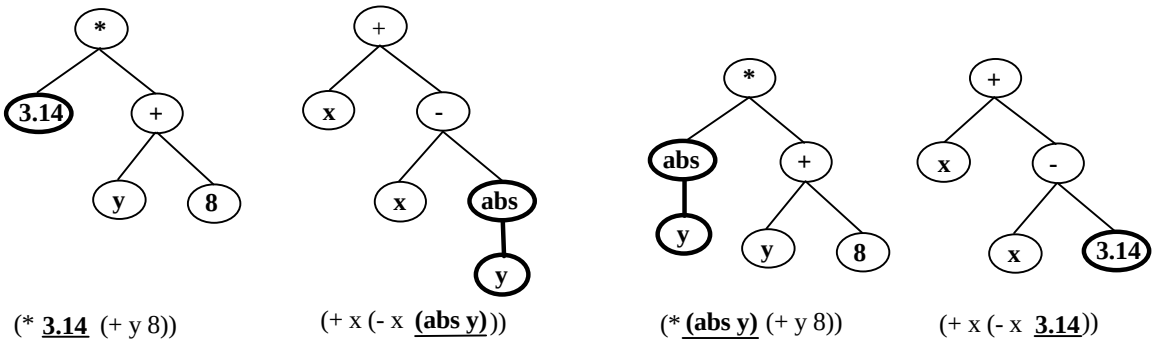


Figure 2: Parse trees (a) before crossover and (b) after crossover. The bold regions denote the subtrees selected for crossover.

Mutation involves choosing one individual, randomly selecting a node in the parse tree, and replacing the node and its subtree with a randomly generated subtree. Permutation involves choosing one individual, randomly selecting a function node, and randomly permuting the arguments to the function. Editing is an S-expression simplification done on a random individual.

Encapsulation involves choosing an individual and selecting a random subtree. That subtree is replaced by a call to a new encapsulated function with no arguments. When the new function is called, the subtree is evaluated. The purpose of encapsulation is to keep potentially useful subtrees from being disrupted by the crossover operation.

The portions of the new population which are created using these various techniques are user-selectable. Koza reports generating 10% of each generation by reproduction and 90% by crossover, and practically never using the other operations.

3. A User's Guide to libgp

We have developed `libgp`, a library written in C++ which implements the genetic programming paradigm. This library may be used as a tool, callable from any C++ program. The implementation is not tied to any particular machine or operating system, and should be portable to any system running a C++ compiler. Genetic Programming often takes a considerable amount of memory and run-time, so it may not be practical to use GP on a small system. This library was developed and tested on Silicon Graphics Indigo and Crimson and Hewlett-Packard 800 series workstations.

3.1. Essential Elements of a GP Run

Koza presents a *tableau* for each problem presented in [9] which summarizes choices which must be made for a GP run. The essential elements of any GP problem are:

- Objective: a statement of the problem which we are attempting to solve.
- Terminal set: the list of terminals which can be included in any S-Expressions generated by the GP.
- Function set: the list of functions which can be included in any S-Expressions generated by the GP.
- Fitness cases: a problem dependent set of test cases which will be used by the fitness function to derive the raw fitness value for each individual.
- Raw fitness: the method by which the fitness function will calculate the raw fitness measure for each individual.
- Standardized fitness: An optional method of conversion between raw fitness and standardized fitness.
- Hits: an auxiliary measure of the number of fitness cases which achieve satisfactory results.
- Wrapper: an optional conversion between output values of an evolved S-Expression and input values needed by the fitness function.
- Parameters: any of the major or minor parameters which control the GP run.

- Success predicate: an optional condition which may terminate the GP run before the maximum number of iterations.

The onus is upon the user of the GP function to decide exactly what values to choose for all of these variables. However, all programs which utilize `libgp` will have similar structure. Figure 3 contains a schematic of what such a program will tend to look like in C++.

```
// We will need certain header files
#include <iostream.h>
#include <math.h>

#include "gp.h"


// This is the fitness function
float fitness_function (S_Expression *s, int *hits)
{
  for (int i = 0;  i < number_of_fitness_cases;  ++i) {
      // Compute each fitness case separately
    }
  *hits = how_many_hits;   // Set the number of hits
  return raw_fitness;      // Return the raw fitness value
}



int main (int argc, char **argv)
{
  // Specify your terminal set

  // Specify your function set

  // Precalculate your set of fitness test cases

  // Declare the GP object
  GP gp (fitness_function, population_size);

  // Specify any non-default GP parameters

  // Run the evolution!
  gp.go (number_of_generations);

  // Examine results
}
```

Figure 3: Schematic Representation of a Program Using `libgp`

3.2. Specifying a Terminal Set

There is a global variable `Tset` included in the `libgp` library. This variable is an instance of the class `TerminalSet`. There are a number of operations which can be performed on this object.

3.2.1. Adding Terminals to the Terminal Set

The most important operation which must be performed prior to starting a GP run is to specify which terminals should be in the set. To add a new terminal to the set, use the member function `add`, passing the name of the terminal in double quotes.

```
Tset.add ("x");
Tset.add ("velocity");
```

In addition, you may initialize a variable with a particular floating point value:

```
Tset.add ("cost", 3.5);
Tset.add ("omega", -1);
```

3.2.2. Using Random Constants

For some applications, it may be desirable for random constants to be included in the terminal set. Koza calls this the *ephemeral random constant*. There is a global function pointer, `ephemeral_constant`, which specifies a function to compute the ephemeral random constant. The default value for this pointer is `NULL`. If you never modify this variable, no ephemeral constant will be used.

In order to add the ephemeral random constant to your terminal set, all you have to do is point this variable to a function which takes no arguments and returns a random single precision floating point value. The `libgp` library includes such a function, called `default_ephemeral_generator`. This generator returns random constants uniformly distributed over the interval [-1,1]. Here is an example of how to specify to use these ephemeral random constants:

```
ephemeral_constant = default_ephemeral_generator;
```

For some applications, you may want a different distribution of random constants. For example, perhaps you want your ephemeral constant to be a positive integer between 1 and 10. Then you could provide your own random number generator:

```
float my_generator (void)
{
  int i = /* A random integer between 1 and 10 */;
  return (float) i;
}
```

Then, to specify that this routine should be used to generate the ephemeral random constants:

```
ephemeral_constant = my_generator;
```

3.2.3. Accessing Terminals

There are times when you need to access or modify the value of a terminal. In particular, your fitness function must set the values of the terminals for each fitness case. This is done primarily with two member routines: `get` and `modify`. The `get` function, given the name of a terminal, will return the current floating point value of that terminal:

```
x_value = Tset.get ("x");
```

This example will retrieve the current value of the terminal "x" and place that value in the floating point variable `x_value`.

The `modify` member function can be used to alter the current value of a terminal. Here is an example, which will set the value of the terminal "x" to the value 3.75:

```
Tset.modify ("x", 3.75);
```

3.3. Specifying a Function Set

There is a global variable `Fset` included in the `libgp` library which is an instance of the class `FunctionSet`.

3.3.1. Specifying a Function Set -- Novice

Before starting a GP run, it is necessary to specify which functions should be used for the run. The `libgp` library has already incorporated several common functions so that you do not have to write them. The routines already provided include basic arithmetic (addition, subtraction, multiplication, protected division), mathematical operations (sine, cosine, exp, etc.), logical operations (and, or, if), and control statements (e.g. progn, iflte).

It is important to note that even though the library contains these functions, the function set **contains no functions by default**. This means that you need to add each function individually if you want it in the function set for a particular run. Each function in the set has its own routine, taking no arguments, which adds it to the function set. The routines should be self-explanatory, and are listed below:

```
use_addition ();              use_subtraction ();
use_multiplication ();        use_protected_division ();
use_sin ();                   use_cos ();
use_exp ();                   use_rlog ();
use_gt ();                    use_abs ();
use_sig ();                   use_atg ();
use_and ();                   use_not ();
use_or ();                    use_if ();
use_eq ();                    use_srt ();
use_sq ();                    use_cub ();
use_progn2 ();                use_progn3 ();
use_ifltz ();                 use_iflte ();
use_ifltz ();                 use_iflte ();
use_srexp ();                 use_pow ();
```

```
use_du ();
```

These routines will do all the work for you to add the particular function to the function set.

3.3.2. Specifying a Function Set -- Advanced

Many problems that you will wish to solve using GP will necessitate their own problem-dependent functions. You can add your own functions to the function set using the member function `add` of the function set variable `Fset`. The `add` member function takes three required parameters and two optional parameters. The required parameters are the name of the function, the number of arguments that it takes, and a pointer to the implementation of the function, respectively.

As an example, the following would declare the `pow` function, which takes two arguments. The function returns the first argument raised to the power of the second argument.

```
Fset.add ("pow", 2, pow_implementation);
```

The first argument is the name of the function, enclosed in double quotes. The function name can include any alphanumeric characters, and/or symbolic characters. We just as easily could have specified "^" as the name of the power function. The second argument is an integer which represents the number of arguments that the function is expected to have. `libgp` does not support functions with variable numbers of arguments. Finally, the third argument is a pointer to a C++ function which implements the function which we are adding to the set. The function must operate in a certain way, which is described in section \ref{subsec-impfun} of this document.

There are also two optional arguments to pass to the `add` function. The first argument is a pointer to a function which performs the *editing* operation on a S-Expression node for this function. If you examine the code in `stdfunc.c` (given in the next chapter of this document), you can get an idea for what editing functions do and how to write them. An editing function is not required, and adding an incorrect editing functions is an especially efficient way to add bugs to your program. The optional fourth argument to `add` should be set to `NULL` if you do not want to specify an editing function.

Finally, the last argument is an integer which specifies whether your user-defined function has side effects. Passing zero (the default) indicates that there are no side effects of calling your function. In other words, this would indicate that your function is a purely mathematical computation, and changes no global variables in your program. If your function changes any non-local variables, you *must* send a nonzero value for this parameter. Otherwise, the editing function (if it is ever performed) may result in an incorrect program.

To illustrate these parameters, the line below adds a new user-defined function which has side-effects, but no editing function:

```
Fset.add ("myfunction", 2, myimplementation, NULL, 1);
```

The fourth parameter, `NULL`, indicates that you are not providing an editing parameter. The fifth parameter, 1, indicates that the function may have side-effects.

3.3.3. Function Implementation

If you choose to add your own function, there is a particular format which your implementation function must follow. As an example, the C++ function below implements the `pow` function that we described above.

```
float pow_implementation (S_Expression **args)
{
  return (float) pow (args[0]->eval(), args[1]->eval());
}
```

This implementation function evaluates its two arguments, and calls the standard C++ `pow` routine to raise the first argument to the power of the second argument.

There are several things to note about implementation functions:

- The implementation function *must* return a `float`.
- The implementation function takes one argument, of type `(S_Expression **)`.
- The `args` parameter points to an array of pointers to `S_Expression` objects. The n^{th} element of this array corresponds to the S-Expression which is the n^{th} argument to the called function.
- Argument i may be evaluated by calling the `eval` member function of that argument. In other words, the function `args[i]->eval()` causes argument i to be evaluated, and returns a floating point value representing its numerical evaluation.
- You need not evaluate every argument. An *if-then-else* type function, for example, should evaluate its first argument, then evaluate one or the other of its second and third arguments, depending on the value returned by its first argument. This is especially important if the arguments contain calls to functions with side effects. In addition, it is inefficient to evaluate arguments which are not needed.
- It is important to evaluate no more arguments than the function expects. It is up to you to ensure that your implementation use exactly the same number of arguments as you specified when use called `Tset.add()`.

Examination of the implementation functions contained in `stdfunc.c` should provide any additional details about user function implementation.

3.4. Specifying GP Parameters

Once you have specified the terminal set and function set, you should declare your GP object and set up the major and minor parameters which control the GP run. A GP object may be declared as follows:

```
GP gp (fitness_function, popsize);
```

This constructor function for an object of class `GP` takes two arguments. The first argument is a pointer to your fitness evaluation function (described in the next section). The second argument is an integer which specifies the size of the population to use for this GPrun.

There are many members of a `GP` object which you can set to control the GP run. Each member has a default value, and may be set with a simple assignment. Below is a summary of the parameters which you may wish to set:

- `float pc:` The probability of crossover (default = 0.9).
- `float pr:` The probability of reproduction (default = 0.1).
- `float pm:` The probability of mutation (default = 0).
- `float pp:` The probability of permutation (default = 0).
- `float pen:` The probability of invoking encapsulation (default = 0). Note that pc + pr + pm + pp + pen should sum to 1.0.
- `float pip:` The probability of internal crossover (must be between 0 and 1, default is 0.9).
- `int Dcreated`: The maximum depth of the any S-Expressions created on any generation (default = 17).
- `int Dinitial:` The maximum depth of the random S-Expressions created on generation 0 (default = 6).
- `int fed:` The frequency of performing the editing operation (default = 0, which means never).
- `CONDITION dec_cond:` A pointer to a `CONDITION` function which gives the conditions under which the decimation operation should be performed (default = NULL, which means never).
- `float pd:` The decimation percentage (default is arbitrarily set to zero, but this parameter is meaningless in the absence of a non-NULL `dec_cond`).

- `generative_method:` specifies the method of generating the initial random S-Expressions. Can be one of the following: `GROW`, `FULL`, or `RAMPED_HALF_AND_HALF`, which correspond to the three methods of generating the initial population described in [9]. The default value is `RAMPED_HALF_AND_HALF`.
- `reproduction_selection:` Specifies the method of choosing individuals for reproduction, and for choosing the first parent for crossover. It can take on one of the following values: `UNIFORM`, `FITNESS_PROPORTIONATE`, `TOURNAMENT`, or `RANK`. The default value is `FITNESS_PROPORTIONATE`.
- `second_parent_selection:` Specifies the method of choosing the second parent for crossover. It can take on the same values as `reproduction_selection`, and has default value `FITNESS_PROPORTIONATE`.
- `int tournament_size:` Specifies the tournament size, if `TOURNAMENT` is the selected method for one of the previous two parameters. Default value is 2, but will not be used unless tournament selection is utilized.
- `int use_greedy_overselection:` Nonzero indicates that greedy overselection should be used (default = 0).
- `float overselection_percentage:` If greedy overselection is used, then 80% of individuals for reproduction or crossover will be chosen from the pool of individuals with `overselection_percentage` portion of the normalized fitness, and the other 20% will be chosen from the remainder of the pool. Default value is 0.32.
- `int use_elitist_strategy:` If nonzero, the best individual from generation n will automatically be placed into generation n+1. Default value is zero.
- `CONDITION termination_criteria:` A pointer to a `CONDITION` function which indicates early termination criteria. The default value is `NULL`, which indicates that the run will only terminate when the proper number of generations has been completed.
- `FLOATFUNC standardize_fitness:` A pointer to a problem-dependent function function which converts raw fitness values to standardized fitness values. The default is `NULL`, which indicates that standardized fitness is equal to raw fitness.

3.5. Specifying a Fitness Function

We mentioned earlier that a pointer to the problem-dependent fitness function must be passed to the GP class constructor. The fitness function must have a particular format. It takes two parameters, an (S_Expression *) which points to the program whose fitness we are testing, and an (int *) into which we should place the number if fitness hits. The return value of a fitness function is a float which returns the raw fitness for this particular individual.

Reproduced below is a sample fitness function. Nfc is the number of fitness cases to be tested. The array xvals[] contains the precalculated values for the variable "x" which represent the different fitness cases to test. The array yvals[] contains the precalculated results which correspond the the "x" values.

```
float fitness_function (S_Expression *s, int *hits)
{
  float total = 0;

  *hits = 0;
  for (int k = 0;  k < Nfc;  ++k) {
      Tset.modify ("x", xvals[k]);
      float y = s->eval();
      float diff = fabs (yvals[k] - y);
      if (diff <= 0.001)
         *hits += 1;
      total += diff;
    }
  return total;
}
```

The fitness function loops over the Nfc fitness cases. For each fitness case, it sets the terminal "x" to the proper value for that fitness case. The S-expression for which we are calculating the fitness is evaluated using its eval() member function, and the result is stored in y. The value of y is compared to the precomputed correct result for that fitness case, yvals[k]. When difference between the computed value and expected value is less than 0.001, the hits counter is incremented. The fitness function returns the raw fitness value, which is the sum of the differences over all fitness test cases.

3.6. Performing a GP Run

Performing a GP run is actually very simple, and consists of the following steps:

1. Adding terminals to the terminal set. This was covered earlier in this section.

2. Adding functions to the function set. This was covered earlier in this section.
3. Precomputing fitness test cases, if necessary.
4. Declaring a `GP` object, specifying the fitness function to be used and the number of individuals in the population. For example,

```
GP gp (fitness_function, 500);
```

5. Specifying any additional parameters to control the GP run. As an example, the code fragment below sets selected parameters:

```
gp.use_greedy_overselection = 1;
gp.termination_criteria = my_termination_criteria;
```

6. Now, you start the run by calling the `go()` member function, passing the maximum number of generations for the run. For example, to start the run with 50 generations,

```
gp.go (50);
```

7. Examine the results.

3.6.1. Verbosity: Keeping Tabs on the Run

The `GP` object has a `verbose` field, which you can set to a number of values. Depending on how you set this field, the GP run may echo information about the run to the terminal (through the `stdout` device). The values are:

- LIST_PARAMETERS echoes all of the parameters for this run to the standard output device.
- TELL_INITIALIZE echoes the message "Creating initial population" when the GP initializes the initial population.
- LIST_INITIAL_EXPRESSIONS causes the initial random S-Expressions to be echoed to the screen as they are created.
- GENERATIONAL_UPDATE causes the current generation number to be echoed to the display, along with the standardized fitness and hits for the best-so-far individual.
- LIST_GENERATIONAL_FITNESSES causes the best individual for each generation to be listed, along with its fitness measures.
- SHOW_EDITED_BEST causes the edited version of the best individual for each generation to be listed (in conjunction with LIST_GENERATIONAL_FITNESSES), or the edited version of the best-of-run individual to be displayed (in conjunction with END_REPORT).

- END_REPORT causes the best-of-run individual to be printed at the end of the run, along with its standardized fitness value, number of hits, and the generation in which it was found.

The default is for none of these to be selected. Each of these values is a single bit value. To select more than one, the `verbose` field should be set by 'or-ing' these values together using the C++ '|' function. For example, to give generation fitness updates and also list each generation's best individual, you should:

```
gp.verbose = GENERATION_UPDATE | LIST_GENERATIONAL_FITNESSES |
      END_REPORT;
```

3.6.2. Examining the Results

After the run is completed, you will probably want to use the best-of-run individual for some other application, or at least print the results. The `GP` object has several member fields which you can examine after the run is completed.

- `Individual best_of_run` is a record of the best individual found during the entire run. This object has several members which are worth examining.
  - `float gp.best_of_run.rfit`, `gp.best_of_run.sfit` are the best individual's raw and standardized fitnesses, respectively.
  - `int gp.best_of_run.hits` represents the number of hits for the best individual of the run.
  - `gp.best_so_far.s` is a pointer to an S-expression for the best program found.

  The S-expression for the best individual may be echoed to an output stream (such as `cout`) using the C++ << operator:

  ```
  cout << "Best so far was:n";
  cout << gp.best_of_run.s << 'n';
  ```

  You can make a copy of the S-expression by using the `copy()` member function:

  ```
  S_Expression *c = gp.best_so_far.s->copy();
  ```

  You can, of course, evaluate the function as necessary by using the `eval()` member function:

  ```
  result = gp.best_so_far.s->eval();
  ```

- `int bestsofar_gen` is an integer recording the generation in which the best-of-run individual was discovered.

Finally, there are a couple more ways to squeeze output from the GP run. By pointing `gp.stat_filename` to a filename, the named file will be written which contains on each line: the generation number, the best standardized fitness, worst standardized fitness, and average standardized fitness for that generation, respectively. For example,

```
gp.stat_filename = "stats.out";
```

3.7. Example 1: The Cart Centering Problem

Koza describes the "cart centering problem" in chapter 7 of [9]. In this problem, we are simulating a cart on a frictionless one-dimensional track. The cart has a random starting position and velocity. The cart has a rocket which emits a constant force (a 'bang-bang force') either right or left. The goal of the problem is to develop a strategy, or formula, which tells us which direction to point the rocket at any particular time in order to center the cart at the origin with a velocity of approximately zero. The reader should refer to [9] for more details.

A program listing is given below which solves this problem using `libgp`. The comments in the code should serve to explain the operation of the program.

```
////////////////////////////////////////////////////////////////////////
//
// testcart.c -- test GP on the "cart centering problem"
//
// written by Larry I. Gritz, 1993
// The George Washington University, Dept. of Elec. Engr & Comp. Science
// Computer Graphics and Animation Lab
//
////////////////////////////////////////////////////////////////////////


#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#include "gp.h"



// Nfc is the number of fitness cases to test
#define Nfc 20

// Tmax is the maximum time that we give any strategy to center the
// cart.  If the cart is not centered by this time, we decide that
// the problem has timed out.
#define Tmax 10.0

// xvals[] and vvals[] contain the x (position) and v (velocity) values,
// respectively, for each of the Nfc precomputed test cases.
float xvals[Nfc];
float vvals[Nfc];
```

```
// Here is our fitness function.  For each of the Nfc fitness cases, we
// run the cart simulation with time step tau = 0.02.  The number of hits
// is the number of test cases for which we successfully center the cart
// without timing out.  The raw fitness is the sum over all test cases
// of the time it took to center the cart.  If the strategy times out,
// its fitness is set to Tmax.
//
float fitness_function (S_Expression *s, int *hits)
{
  const float tau = 0.02;       // time step used for the simulation
  float total = 0;              // total raw fitness

  *hits = 0;
  for (int k = 0;  k < Nfc;  ++k) {
      float x = xvals[k];             // get position for this test case
      float v = vvals[k];             // get velocity for this test case
      for (float t = 0.0;  t < Tmax;  t += tau) {
          Tset.modify ("x", x);
          Tset.modify ("v", v);

          // Acceleration is given by evaluating the S-expression.  If the
          // evaluated S-expression is greater than zero, acceleration is
          // +1, else acceleration is -1.

          float a = 0.5 * (s->eval() > 0.0 ? 1.0 : -1.0);
          // Use Euler integration to determine next x and v
          x += tau * v;
          v += tau * a;
          // Terminate if we're at the origin and not moving
          if (fabs (x) < 0.05 && fabs (v) < 0.05) {
              *hits += 1;
              break;
            }
        }
      // If we timed out, set t = 10.0
      if (t > 10.0)
          t = 10.0;
      total += t;
    }
  return total;
}



// For standardized fitness, we divide the raw fitness by the number
// of fitness cases.  (This is unnecessary, but makes output look a little
// nicer.
//
float standardize_fitness (float r)
{
  return r / Nfc;
}



int main (int argc, char **argv)
{
  int popsize = 500;
  int ngens = 50;

  // If there are three arguments on the command line, use the first
  // two for an alternate population size and number of generations,
  // respectively, and the third argument as a new seed value for the
  // random number generator.
  //
  if (argc > 2) {
      popsize = atoi (argv[1]);
      ngens = atoi (argv[2]);
      cout << "Running " << popsize << " individuals for " << ngens
           << " generations\n\n";
    }
  if (argc > 3)
      seed_random (atoi (argv[3]));
```

```
  // Add three terminals to the terminal set:  "x", "v", and the numeral "-1"
  //
  Tset.add ("x");
  Tset.add ("v");
  Tset.add ("-1", -1);

  // Use the following functions from libgp's standard function set:
  // +, -, *, %, gt, abs
  //
  use_addition();  use_subtraction();
  use_multiplication();  use_protected_division();
  use_gt();  use_abs();

  // Choose random x and v values for the Nfc fitness test cases.  Choose
  // each value to be uniformly distributed on [-.75, .75].
  //
  for (int i = 0;  i < Nfc;  ++i) {
      xvals[i] = 1.5 * random() - 0.75;
      vvals[i] = 1.5 * random() - 0.75;
    }

  // Declare the GP object with the given fitness function and population
  // size, and set up some run parameters.
  //
  GP gp (fitness_function, popsize);
  gp.verbose = (GPVerbosity)(TELL_INITIALIZE | GENERATION_UPDATE |
                             END_REPORT);
  gp.standardize_fitness = standardize_fitness;

  // Run the genetic simulation!
  gp.go (ngens);

  // Now output the best individual S-Expression to the file "best.out".
  // Three other tricks:  look below to see how we use copy() to copy
  // S-Expressions, edit() to replace them with edited versions, and
  // write() to place ASCII versions of the LISP S-Expressions into
  // a character string.

  S_Expression *c = gp.best_so_far.s->copy();
  char buffer[2000];
  c = edit (c);
  c->write (buffer);
  FILE *file = fopen ("best.out", "w");
  fprintf (file, "%s\n", buffer);
  fclose (file);
  return 0;
}
```

3.8. Example 2: The 6-Multiplexor Problem

There is considerable discussion in [9] of what Koza calls the "6-Multiplexor" problem. This problem involves finding a formula for a 6-multiplexor with four data lines and 2 address lines. The address lines determine which of the four data lines should be designated as the output of the multiplexor. Below is a fairly straightforward implementation of this problem using `libgp`.

```
//////////////////////////////////////////////////////////////////////
//
// test6m.c -- test GP on the "6-multiplexor" problem
//
// written by Larry I. Gritz, 1993
// The George Washington University, Dept. of Elec. Engr & Comp. Science
// Computer Graphics and Animation Lab
//
//////////////////////////////////////////////////////////////////////
```

```
#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#include "gp.h"
#include "random.h"



// Our fitness function loops over the 64 possible cases for the
// 6-multiplexor (since 2^6 = 64).  The address is the high two bits,
// and the data are the lower 4 bits.  We set the 6 relevant variables,
// then run the S-expression for this individual.  Any nonzero value
// is considered a logical one.  The number of its is the number of
// correct matches.  The raw fitness is the number of misses.
//
float fitness_function (S_Expression *s, int *hits)
{
  *hits = 0;
  for (int k = 0;  k < 64;  ++k) {
      int a = (k >> 4);
      int correct = (k & (1 << a)) ? 1 : 0;
      Tset.modify ("a0", (a&1) ? 1 : 0);
      Tset.modify ("a1", (a&2) ? 1 : 0);
      Tset.modify ("d0", (k&1) ? 1 : 0);
      Tset.modify ("d1", (k&2) ? 1 : 0);
      Tset.modify ("d2", (k&4) ? 1 : 0);
      Tset.modify ("d3", (k&8) ? 1 : 0);
      int calculated = (s->eval() > 0);
      if (correct == calculated)
          *hits += 1;
    }
  return 64 - *hits;
}



// Termination criteria: if we come across a case that has 64 hits, then
// it is a 100% correct solution and we do not need to run any more trials.
//
int termination_criteria (GP *gp)
{
  if (gp->best_of_run.hits == 64)
      return 1;
  else return 0;
}



int
main (int argc, char **argv)
{
  int popsize = 500;
  int ngens = 50;
  int nruns = 100;

  // If there are three arguments on the command line, use the first
  // two for an alternate population size and number of generations,
  // respectively, and the third argument as a new seed value for the
  // random number generator.
  //
  if (argc > 2) {
      popsize = atoi (argv[1]);
      ngens = atoi (argv[2]);
      cout << "Running " << popsize << " individuals for " << ngens
           << " generations\n\n";
    }
  if (argc > 3)
      seed_random (atoi (argv[3]));

  // Add the two address and four data values to the terminal set.
  Tset.add ("a0");
  Tset.add ("a1");
```

```
  Tset.add ("d0");
  Tset.add ("d1");
  Tset.add ("d2");
  Tset.add ("d3");

  // Use the canned logical AND, OR, NOT, and IF functions
  use_and();
  use_or();
  use_not();
  use_if();

  // Set up the GP object and let it run
  //
  GP gp (fitness_function, popsize);
  gp.verbose = (GPVerbosity) (END_REPORT | SHOW_EDITED_BEST);
  gp.termination_criteria = termination_criteria;
  gp.go (ngens);

  cout << "\nDone.\n";
  cout.flush();
  return 0;
}
```

4. libgp Source Code

The previous chapter should have made it reasonably clear how the GP library is organized. This chapter contains source code listings for a C++ implementation of GP which has approximately the same functionality as the LISP version given in Koza's book.

C++ classes are given for terminal and function sets, S-Expressions, Individuals, and GP objects. Perusing through the first code listing of `gp.h`, which contains the class definitions for `libgp`, should give the reader any additional documentation needed on the use of the library. Source code listings of the implementations of these classes follow.

The reader of this technical report should feel welcome to use this code, in part or in whole, to experiment with GP or to get ideas for his or her own GP implementations. Appropriate citations to this document, when ideas or code contained here are used, would be appreciated.

4.1. gp.h --- Class definitions

```
//////////////////////////////////////////////////////////
// gp.h -- class definitions for Genetic Programming in C++
//
// written by Larry I. Gritz, 1993
// The George Washington University, Dept. of EE&CS
// Computer Graphics and Animation Lab
//////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////
// Some type declarations and forward declarations
//////////////////////////////////////////////////////////

class S_Expression;
class GP;

typedef float (*impfunc)(S_Expression **);
typedef S_Expression* (*editfunc)(S_Expression *);
typedef int   (*CONDITION)(GP *);      // A condition
                                       // function pointer
typedef float (*FLOATFUNC)(float);     // takes float,
                                       // returns float

// takes nothing, returns float
typedef float (*EPHEMERAL)(void);

// Types of S_Expression nodes
enum SEXP_TYPE {STnone, STconstant, STterminal, STfunction};

// Ways of generating random S-Expression trees
enum GenerativeMethod { GROW, FULL, RAMPED_HALF_AND_HALF };

// Maximum number of arguments to an S-Expression function
#define MAX_SEXP_ARGS 4

// Fitness evaluation function
typedef float (*FITNESSFUNC)(S_Expression *s, int *hits);

// Methods of selecting individuals for reproduction
enum SelectionMethod { UNIFORM, FITNESS_PROPORTIONATE,
                       TOURNAMENT, RANK };

// Types of fitness measures
enum FitnessMeasure { RAW, ADJUSTED };

// Flags for how much stuff to print out during a run
enum GPVerbosity { QUIET = 0,
                   TELL_INITIALIZE = 1,
                   LIST_INITIAL_EXPRESSIONS = 2,
                   GENERATION_UPDATE = 4,
                   LIST_GENERATIONAL_FITNESSES = 8,
                   SHOW_EDITED_BEST = 16,
                   END_REPORT = 32,
                   LIST_PARAMETERS = 64,
                   DEBUG = -1
                 };



//////////////////////////////////////////////////////////
// TerminalSet class
//////////////////////////////////////////////////////////

class TerminalSet {

 private:

  struct Terminal {
      char  *name;       // Name of the terminal
      float  val;        // Current value
    };

  int       maxn;        // Total terminals allocated
  Terminal *terminals;   // The actual terminals

 public:
  int       n;           // Current number of terminals



 public:

  // Constructor and destructor
  TerminalSet (void);
  ~TerminalSet (void);

  // Add a new terminal to the set
  void add (const char *name, float val = 0);

  // Modify the value of a terminal specified by name
  void modify (const char *name, float val);

  // Modify the value of a terminal specified by index
  void modify (const int index, float val)
         { terminals[index].val = val; }

  // Retrieve the value of a terminal specified by name
```

```
  float get (const char *name);

  // Look up the value based on the index number
  float get (int ind) { return terminals[ind].val; }

  // Return the index number for the named terminal
  int index (const char *name);

  // Look up the value based on the index number
  float lookup (int ind) { return terminals[ind].val; }

  // Look up the name based on the index number
  char *getname (int ind) { return terminals[ind].name; }

  // Print entire table
  void print (void);

};


// Definition for the terminal set
extern TerminalSet Tset;



//////////////////////////////////////////////////////
// FunctionSet class
//////////////////////////////////////////////////////

class FunctionSet {

 private:
  struct SFunction {
      char      *name;      // Name of the function
      int        nargs;     // Number of arguments it takes
      impfunc    func;      // The implementation
      editfunc   edit;      // The editing function
      int        active;    // 0 == don't use this function
      int        side_effects;   // 1 == has side effects
      S_Expression *s;      // non-NULL means that it's
                            // encapsulated function.
    };

  int        maxn;           // Total functions allocated
  SFunction *functions;      // The actual function records
  int        nencapsulated;  // Number of encaps. functions
 public:
  int         n;             // Current number of functions

  // Constructor & Destructor
  FunctionSet (void);
  ~FunctionSet (void);
```

```
// Add a new function to the set
void add (const char *name, int nargs,
          impfunc implementation,
          editfunc edit_function = NULL,
          int has_sides = 0);

// Add a new function which executes an S-Expression
int encapsulate (S_Expression *s, int nargs = 0);

// Remove a named function from the set
void remove (const char *name) {
    int i = index (name);
    if (i >= 0)
        functions[i].active = 0;
  }

// Remove an indexed function from the set
void remove (int ind) {
    if (ind >= 0  ||  ind < n)
        functions[ind].active = 0;
  }

// Return the index number for the named function
int index (const char *name);

// Look up the name of a function based on the index
char *getname (int ind);

// Look up the # of arguments of a function
int nargs (int ind) { return functions[ind].nargs; }

// Return 1 if the indexed function has side effects
int has_sideeffects (int ind)
      { return functions[ind].side_effects; }

// Return a pointer to indexed function implementation
impfunc lookup_implementation (int ind)
      { return functions[ind].func; }

// Return a pointer to the indexed function encapsulation
S_Expression *lookup_encapsulation (int ind)
      { return functions[ind].s; }

// Return 1 if the indexed function is an encapsulation
int is_encapsulated (int ind)
      { return (functions[ind].s ? 1 : 0); }

// Return a pointer to indexed function edit operation
editfunc editop (int ind) { return functions[ind].edit; }

// Print entire table
void print (void);
```

```
};


// Declare the globally visible function set
extern FunctionSet Fset;



////////////////////////////////////////////////////////
// S_Expression class
////////////////////////////////////////////////////////

class S_Expression {

 public:
  SEXP_TYPE  type;
  float      val;      // value if type == STconstant
  int        which;    // index of terminal or function
  S_Expression *args[MAX_SEXP_ARGS];

  // Constructor
  S_Expression (void);

  // new and delete operators
  void* operator new (size_t size);
  void  operator delete (void *);

  // Runs an encapsulated program, specified by index
  friend float run_encapsulated_program (int ind);

  // Evaluate this S_Expression
  float eval (void)
  {
    float f;
    if (type == STfunction) {
        if (Fset.is_encapsulated (which))
            f = run_encapsulated_program (which);
        else {
            impfunc func =
                (impfunc) Fset.lookup_implementation (which);
            f = (*func)(args);
          }
      }
    else if (type == STterminal)  f = Tset.lookup (which);
    else if (type == STconstant)  f = val;
    else {
        cerr << "Error: bad case in S_Expression::eval\n";
        char buffer[2048];
        write (buffer);
        cerr << buffer << '\n';
        f = 0;
      }
    return f;
  }

  // Edit the tree (destructively), return ptr to new root
  friend S_Expression *edit (S_Expression *s);

  // Permute the arguments of this functional S-Expression
  void permute (void);

  // Make another copy
  S_Expression *copy (void);

  // Test for equivalence between two S_Expressions
  friend int equiv (S_Expression *s1, S_Expression *s2);

  // Is the S_Expression a numerical constant?
  int is_numerical (float& f);

  // Count stuff
  void characterize (int *depth, int *total, int *internal,
                     int *external);

  // Does the tree below have functions with side effects?
  int side_effects (void);

  // Select points
  S_Expression * selectany (int, int *, S_Expression ***ptr);
  S_Expression * selectinternal (int, int *, S_Expression ***ptr);
  S_Expression * selectexternal (int, int *, S_Expression ***ptr);
  S_Expression * select (float pip, S_Expression ***ptr);

  // Perform crossover operation
  friend void crossover (S_Expression **s1,
                         S_Expression **s2, float pip);

  // Make a random tree
  friend S_Expression *random_sexpression (
    GenerativeMethod strategy, int maxdepth=6, int depth=0);

  // Chop off below a certain depth
  void restrict_depth (int maxdepth = 17, int depth = 0);

  // Write the lisp code into a string
  void write (char *, int level = 0);

  // Output to a stream
  friend ostream& operator<< (ostream& out, S_Expression *s);

  // Make an S_Expression from a LISP string
  friend S_Expression *sexify (char **s);
  friend S_Expression *sexify (char *s);
};
```

```
extern EPHEMERAL ephemeral_constant;




////////////////////////////////////////////////////////
// Individual and GP classes
////////////////////////////////////////////////////////

class Individual {

 public:
  S_Expression *s;       // The S-expression
  float  rfit, sfit;     // raw, standardized, adjusted, and
  float  afit, nfit;     //    normalized fitness measures
  float  sumnfit;        // Sum of nfitnesses up to this guy
  int    hits;           // Number of hits
  int    recalc_needed;  // Do we need to recalculate?

  // Constructor and destructor
  Individual (void) { s = NULL;  recalc_needed = 1; }
  ~Individual (void) { if (s) delete s; }

  // Assignment operator
  void operator= (Individual& i) {
      if (s)    delete s;
      if (i.s)  s = i.s->copy();
      else      s = NULL;
      rfit = i.rfit;
      sfit = i.sfit;
      afit = i.afit;
      nfit = i.nfit;
      sumnfit = i.sumnfit;
      hits = i.hits;
      recalc_needed = i.recalc_needed;
    }
};



class GP {

 public:
 // Parameters controlling the genetic programming run
   int M;            // The population size
   int G;            // The number of generations to run
   float pc;         // Probability of crossover
   float pr;         // Probability of reproduction
   float pip;        // Probability of internal crossover
   int   Dcreated;   // Max depth of S-Expressions
   int   Dinitial;   // Max depth of initial population
   float pm;         // Probability of mutation
   float pp;         // Probability of permutation
   int   fed;        // Frequency of performing editing
   float pen;        // Probability of encapsulation
   CONDITION dec_cond; // Condition for decimation
   float pd;         // Decimation percentage

   GenerativeMethod generative_method;
   SelectionMethod reproduction_selection;
   SelectionMethod second_parent_selection;
   int use_greedy_overselection;
   float overselection_boundary;
   int use_elitist_strategy;
// Number to use when using tournament selection
   int tournament_size;

// Housekeeping information
   int  initialized;
   int  gen;             // Current generation number
   Individual *pop;            // The actual population
   Individual *newpop;         // Population we're building
   Individual  best_of_run;    // Housekeeping information
   int   bestofrun_gen;   // Gen when best_so_far was found
   int   bestofgen_index; // index of this generation's best
   int   bestofgen_hits;  // hits of this generation's best
   float bestofgen_sfit;  // sfitness of this gen's best
   float worstofgen_sfit; // sfitness of this gen's worst
   float avgofgen_sfit;   // average sfitness of this gen

 // User-defined funcs for controlling the GP run and I/O
   int  verbose;
   FITNESSFUNC fitness_function;   // The fitness evaluation
function
   CONDITION termination_criteria; // When do we terminate?
   FLOATFUNC standardize_fitness; // Fitness standardization
   float sfit_dontreport;    // Don't report fitness >= this
   CONDITION generation_callback;  // Call me after each gen
   int bestworst_freq;    // How often to report best/worst?
   char *stat_filename;
   FILE *stat_file;


 // Public methods
 public:

  // Constructor & destructor
  GP (FITNESSFUNC fitness_function, int popsize = 500);
  ~GP (void);

  // The actual GP run is invoked with "go":
  void go (int maxgens = 50);
```

```
  // Print all the S-expressions and fitness values
  void print_population (void);

 // Stuff used internally
 private:

  // Initialize various structures
  void init (void);

  // Create the initial random population
  void create_population (void);

  // Make a new generation from the current one
  void nextgen (void);

  // Sort pop by normalized fitness
  void sort_fitness (void);

  // Calculate fitnesses & stats
  void eval_fitnesses (void);

  // Print some end-of-run statistics
  void report_on_run (void);

  // List the parameters for the run
  void list_parameters (void);
};


float default_ephemeral_generator (void);



////////////////////////////////////////////////////////
// "Novice" canned declarations for commonly used functions
// found in stdfunc.c.
////////////////////////////////////////////////////////

void use_addition (void), use_subtraction (void);
void use_multiplication (void), use_protected_division (void);
void use_sin (void), use_cos (void), use_atg (void);
void use_exp (void), use_rlog (void);
void use_gt (void), use_abs (void), use_sig (void);
void use_and (void), use_or (void), use_not (void);
void use_if (void), use_eq (void);
void use_progn2 (void), use_progn3 (void);
void use_srt (void), use_sq (void), use_cub (void);
void use_ifltz (void), use_iflte (void);
void use_srexpt (void), use_pow (void);
void use_du (void);
extern int max_du_iterations;
void use_setsv (void);
```

4.2. terminal.c --- Terminal set implementation

```
////////////////////////////////////////////////////////
// terminal.c - implementation for S-Expression terminals
//
// written by Larry I. Gritz, 1993
// The George Washington University, Dept. of EE&CS
// Computer Graphics and Animation Lab
////////////////////////////////////////////////////////

#include <iostream.h>
#include <malloc.h>
#include <string.h>
#include <stdio.h>
#include "gp.h"


// Constructor for a TerminalSet
TerminalSet::TerminalSet (void)
{
  n = 0;  // Currently no terminals in the list
  maxn = 16;  // Allocate room for 16 terminals to start
  terminals = (Terminal *) malloc (maxn*sizeof(Terminal));
}



// Destructor
TerminalSet::~TerminalSet (void)
{
  if (terminals) {
      for (int i = 0;  i < n;  ++i)
          free (terminals[i].name);
      free (terminals);
    }
}



// Add a new terminal to the set
void TerminalSet::add (const char *name, float val)
{
  // First, check to see if the name is already in the list
  for (int i = 0;  i < n;  ++i)
```

```
      if (! strcmp (terminals[i].name, name)) {
          cerr << "TerminalSet Error: attempt to add existing
terminal: "
               << terminals[i].name << '\n';
          return;
        }
  // If we need more space, allocate it
  if (n == maxn - 1) {
      maxn *= 2;
      terminals = (Terminal *) realloc (terminals,
                                maxn * sizeof (Terminal));
    }
  // Okay, now add it
  terminals[n].name = strdup (name);
  terminals[n++].val = val;
}



// Modify the value of a terminal in the set
//
void TerminalSet::modify (const char *name, float val)
{
  for (int i = 0;  i < n;  ++i)
      if (! strcmp (terminals[i].name, name)) {
          terminals[i].val = val;
          return;
        }
  // Oops!  We didn't find the terminal
  cerr << "TerminalSet Error: attempt to modify non-existing
terminal: "
       << name << '\n';
}



// Retrieve the current value of a terminal
float TerminalSet::get (const char *name)
{
  for (int i = 0;  i < n;  ++i)
      if (! strcmp (terminals[i].name, name))
          return terminals[i].val;
  // Oops!  We didn't find the terminal
  cerr << "TerminalSet Error: could not find terminal: " <<
name << '\n';
  return 0;
}



// Return the index number for a named terminal
int TerminalSet::index (const char *name)
{
  for (int i = 0;  i < n;  ++i)
      if (! strcmp (terminals[i].name, name))
          return i;
  // Oops!  We didn't find the terminal
  cerr << "TerminalSet Error: could not index terminal: " <<
name << '\n';
  return -1;
}



// Print the entire table of terminals
void TerminalSet::print (void)
{
  cout << "\nTerminal set listing:";
  cout << "\n---------------------\n";
  for (int i = 0;  i < n;  ++i)
      cout << '\"' << terminals[i].name << "\" = " <<
terminals[i].val << '\n';
  cout << '\n';
}



// Declaration for the globally visible terminal set
TerminalSet Tset;
```

4.3. func.c --- Function set implementation

```
////////////////////////////////////////////////////////////
// func.c -- class implementation for S-Expression functions
//
// written by Larry I. Gritz, 1993
// The George Washington University, Dept. of EE&CS
// Computer Graphics and Animation Lab
////////////////////////////////////////////////////////////

#include <iostream.h>
#include <malloc.h>
#include <string.h>
#include <stdio.h>
#include "gp.h"
```

```
// Constructor
FunctionSet::FunctionSet (void)
{
  n = 0;  // Currently no functions in the list
  maxn = 16;  // Allocate room for 16 terminals to start
  functions = (SFunction *) malloc (maxn*sizeof(SFunction));
  nencapsulated = 0;  // No encapsulated functions
}



// Destructor
FunctionSet::~FunctionSet (void)
{
  if (functions) {
      for (int i = 0;  i < n;  ++i)
          free (functions[i].name);
      free (functions);
    }
}



// Add a new function to the set
//
void FunctionSet::add (const char *name, int nargs,
                       impfunc implementation,
                         editfunc edit_function,
                       int has_sides)
{
  // First, check to see if the name is already in the list
  for (int i = 0;  i < n;  ++i)
      if (! strcmp (functions[i].name, name)) {
          cerr << "FunctionSet Error: attempt to add existing
function: "
               << functions[i].name << '\n';
          return;
        }
  // Check to see if nargs is in range
  if (nargs > MAX_SEXP_ARGS) {
      cerr << "Error! Function " << name
           << " declared with " << nargs
           << "arguments.\nYou have to change MAX_SEXP_ARGS and
recompile.\n";
    }
  // If we need more space, allocate it
  if (n == maxn - 1) {
      maxn *= 2;
      functions = (SFunction *) realloc (functions,
                                  maxn * sizeof (SFunction));
    }
  // Okay, now add it
  functions[n].name = strdup (name);
  functions[n].nargs = nargs;
  functions[n].func = implementation;
  functions[n].edit = edit_function;
  functions[n].active = 1;
  functions[n].side_effects = has_sides;
  functions[n++].s = NULL;
}



// Add a new function to the set
int FunctionSet::encapsulate (S_Expression *s, int nargs)
{
  // Construct a name
  char buffer[16];
  sprintf (buffer, "(E%d)", nencapsulated);
  ++nencapsulated;

  // If we need more space, allocate it
  if (n == maxn - 1) {
      maxn *= 2;
      functions = (SFunction *)
            realloc (functions, maxn * sizeof (SFunction));
    }
  for (int i = 0;  i < n;  ++i) {
      if (functions[i].s  &&  equiv (s, functions[i].s))
          return i;
    }
  // Okay, it's not a duplicate, so we add it
  functions[n].name = strdup (buffer);
  functions[n].nargs = nargs;
  functions[n].func = NULL;
  functions[n].edit = NULL;
  functions[n].active = 1;
  functions[n].side_effects = s->side_effects();
  functions[n].s = s->copy();
  cerr << "\nEncapsulating " << buffer << " = " <<
functions[n].s << '\n';
  cerr.flush();
  n++;
  return (n-1);
}



// Return the index number for the named function
int FunctionSet::index (const char *name)
{
  for (int i = 0;  i < n;  ++i)
      if (! strcmp (functions[i].name, name))
          return i;
  // Oops!  We didn't find the function
```

```
  cerr << "FunctionSet Error: could not index function: " <<
name << '\n';
  return -1;
}


// Look up the name of a function based on the index number
char * FunctionSet::getname (int ind)
{
  if (ind < 0  ||  ind >= n) {
      cerr << "FunctionSet Error: attempted to getname with bad
index ("
           << ind << ")\n";
      return NULL;
    }
  else return functions[ind].name;
}



// Print entire table
void FunctionSet::print (void)
{
  cout << "\nFunction set listing:\n"
       <<   "---------------------\n";
  for (int i = 0;  i < n;  ++i) {
      cout << '\"' << functions[i].name << "\":  "
           << functions[i].nargs << " parameters ";
      if (! functions[i].active)
          cout << " (deleted) ";
      if (functions[i].s)
          cout << functions[i].s;
      cout << '\n';
    }
  cout << '\n';
}



// Declaration for the globally visible function set
FunctionSet Fset;
```

4.4. sexp.c -- S-Expression implementation

```
////////////////////////////////////////////////////////////
```

```
// sexp.c -- class implementation for S-Expression in C++
//
// written by Larry I. Gritz, 1993
// The George Washington University, Dept. of EE&CS
// Computer Graphics and Animation Lab
////////////////////////////////////////////////////////////

#include <iostream.h>
#include <malloc.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "gp.h"
#include "random.h"


static S_Expression *free_list = NULL;


// Constructor
S_Expression::S_Expression (void)
{
  type = STnone;
  val = 0;
  which = 0;
  for (int i = 0;  i < MAX_SEXP_ARGS;  ++i)
      args[i] = NULL;
}


void* S_Expression::operator new (size_t size)
{
  static long retrieved = 0;
  static long numalloced = 0;
  S_Expression *s;

  if (free_list) {
      s = free_list;
      free_list = s->args[0];
      ++retrieved;
    }
  else {
      s = (S_Expression *) malloc (sizeof (S_Expression));
      ++numalloced;
      if (! (numalloced % 100000))
          cerr << "\nAllocated " << numalloced
               << " sexp cells (" << retrieved << ")\n";
    }
  return s;
}
```

```
void S_Expression::operator delete (void *s)
{
  S_Expression *se = (S_Expression *) s;

  for (int i = 0;  i < MAX_SEXP_ARGS;  ++i)
      if (se->args[i]) delete se->args[i];
  ((S_Expression *)s)->args[0] = free_list;
  free_list = (S_Expression *)s;
}



// Execute an encapsulated program, specified by index.
// This takes the recursion out of S_Expression::eval(),
// so that we can inline it.
float run_encapsulated_program (int ind)
{
  S_Expression *s = Fset.lookup_encapsulation (ind);
  return s->eval();
}



// Perform editing operation on this S_Expression,
// then return new expr
S_Expression *edit (S_Expression *s)
{
  if (s  &&  (s->type == STfunction)) {
      for (int i = 0;  i < Fset.nargs (s->which);  ++i)
          s->args[i] = edit (s->args[i]);
      editfunc e = Fset.editop (s->which);
      return e ? (*e)(s) : s;
    }
  return s;
}



// Perform permutation on a random node in this S-expression
void S_Expression::permute (void)
{
  cout << "Permuting:\nold = " << this << "\nnew = ";
  cout.flush();
  if (type == STfunction  &&  Fset.nargs(which) > 1) {
      for (int i = Fset.nargs(which)-1;  i;  --i) {
          float u = random();
          int k = (int) floor (i * u);
          S_Expression *temp = args[k];
          args[k] = args[i];
          args[i] = temp;
        }
    }
  cout << this << "\n";
  cout.flush();
}



// Return a duplicate copy of the given S-Expression
S_Expression * S_Expression::copy (void)
{
  S_Expression *se = new S_Expression;
  se->type = type;
  se->val = val;
  se->which = which;
  for (int i = 0;  i < MAX_SEXP_ARGS;  ++i)
      if (args[i])  se->args[i] = args[i]->copy();
      else          se->args[i] = NULL;
  return se;
}



// Test for equivalence, return 1 if S-expressions are same
int equiv (S_Expression *s1, S_Expression *s2)
{
  if (! s1 && ! s2)            return 1;
  if (!s1 || !s2)              return 0;
  if (s1->type != s2->type)    return 0;
  switch (s1->type) {
      case STnone :
           cerr << "Error: bad case in S_Expression::==\n";
          return 0;
      case STconstant : return (s1->val == s2->val);
      case STterminal : return (s1->which == s2->which);
      case STfunction :
          if (s1->which != s2->which)
              return 0;
          for (int i = 0;  i < MAX_SEXP_ARGS;  ++i)
              if (! equiv (s1->args[i], s2->args[i]))
                  return 0;
          return 1;
    }
}



// If the S_Expression is a numerical constant, put it in f.
int S_Expression::is_numerical (float& f)
{
  if (type == STconstant) {
      f = val;
      return 1;
    }
  else if (type == STterminal) {
```

```
      char *name = Tset.getname (which);
      if ((name[0] >= '0' && name[0] <= '9') ||
                name[0] == '-') {
          f = Tset.lookup (which);
          return 1;
        }
    }
  return 0;
}



// Characterize the tree, returning its depth, the total
// number of nodesin the tree, the total number of internal
// nodes (those with children), and the total number of
// external nodes (those without children).
void S_Expression::characterize (int *depth,
              int *totalnodes, int *internal, int *external)
{
  *depth = 1;  *totalnodes = 1;
  if (type == STfunction  &&  Fset.nargs(which) >= 1) {
      int child_depth, child_total, child_internal;
      int child_external;
      *internal = 1;
      *external = 0;
      int nargs = Fset.nargs (which);
      int max_child_depth = 0;
      for (int i = 0;  i < nargs;  ++i) {
          args[i]->characterize (&child_depth, &child_total,
                          &child_internal, &child_external);
          *totalnodes += child_total;
          *internal += child_internal;
          *external += child_external;
          if (child_depth > max_child_depth)
              max_child_depth = child_depth;
        }
      *depth += max_child_depth;
    }
  else {
      *internal = 0;
      *external = 1;
    }
}



// Return if the tree contains functions with side effects
int S_Expression::side_effects (void)
{
  if (type == STfunction) {
      if (Fset.has_sideeffects (which))
          return 1;
      for (int i = 0;  i < Fset.nargs (which);  ++i)
          if (args[i]->side_effects())
              return 1;
    }
  return 0;
}



S_Expression * S_Expression::selectany (int m, int *n,
                                        S_Expression ***ptr)
{
  (*n)++;
  if (*n == m)
      return (this);
  if (type == STfunction) {
      int nargs = Fset.nargs (which);
      for (int i = 0;  i < nargs;  ++i) {
          S_Expression *s = args[i]->selectany (m, n, ptr);
          if (s) {
              if (args[i] == s)
                  *ptr = &(args[i]);
              return s;
            }
        }
    }
  return NULL;
}



S_Expression* S_Expression::selectexternal (int m, int *n,
                                            S_Expression ***ptr)
{
  if (type == STfunction  &&  Fset.nargs(which) >= 1) {
      int nargs = Fset.nargs (which);
      for (int i = 0;  i < nargs;  ++i) {
          S_Expression *s=args[i]->selectexternal (m,n,ptr);
          if (s) {
              if (args[i] == s)
                  *ptr = &(args[i]);
              return s;
            }
        }
      return NULL;
    }
  else {
      (*n)++;
      if (*n == m)  return (this);
      else          return NULL;
    }
}
```

```
S_Expression* S_Expression::selectinternal (int m, int *n,
                                            S_Expression ***ptr)
{
  if (type == STfunction  &&  Fset.nargs(which) >= 1) {
      (*n)++;
      if (*n == m)
          return (this);
      int nargs = Fset.nargs (which);
      for (int i = 0;  i < nargs;  ++i) {
          S_Expression *s=args[i]->selectinternal (m,n,ptr);
          if (s) {
              if (args[i] == s)
                  *ptr = &(args[i]);
              return s;
            }
        }
    }
  return NULL;
}



S_Expression * S_Expression::select (float pip,
                                     S_Expression ***ptr)
{
  int depth, total, internal, external;
  int n = -1;
  int which;

  *ptr = NULL;
  characterize (&depth, &total, &internal, &external);
  if (random() < pip && internal >= 1) {
      which = (int) floor (random() * internal);
      return selectinternal (which, &n, ptr);
    }
  else if (external >= 1) {
      which = (int) floor (random() * external);
      return selectexternal (which, &n, ptr);
    }
}



// Perform the crossover between these two S-Expressions
void crossover (S_Expression **s1, S_Expression **s2,
                float pip)
{
  S_Expression **parent1ptr = NULL, **parent2ptr = NULL;
  S_Expression *fragment1, *fragment2;
  char buffer[1024];

  fragment1 = (*s1)->select (pip, &parent1ptr);
  if (! parent1ptr)
      parent1ptr = s1;
  fragment2 = (*s2)->select (pip, &parent2ptr);
  if (! parent2ptr)
      parent2ptr = s2;
  cout.flush();
  *parent1ptr = fragment2;
  *parent2ptr = fragment1;
}



// Choose a random terminal (possibly including the
// ephemeral constant
static void random_terminal (S_Expression *s)
{
  int i = Tset.n + (ephemeral_constant ? 1 : 0);
  s->which = (int) floor (i * random());
  if (s->which >= Tset.n) {
      s->type = STconstant;
      s->val = ephemeral_constant();
    }
  else s->type = STterminal;
}



// Choose a random function
static void random_function (S_Expression *s)
{
  s->type = STfunction;
  s->which = (int) floor (Fset.n * random());
}



// Choose a random terminal or function
static void random_terminal_or_function (S_Expression *s)
{
  int i = Tset.n + Fset.n + (ephemeral_constant ? 1 : 0);
  s->which = (int) floor (i * random());
  if (s->which < Fset.n)
      s->type = STfunction;
  else {
      s->which -= Fset.n;
      if (s->which >= Tset.n) {
          s->type = STconstant;
          s->val = ephemeral_constant();
        }
      else s->type = STterminal;
    }
}
```

```
// Create a random S-Expression
S_Expression *random_sexpression (GenerativeMethod strategy,
                                  int maxdepth, int depth)
{
  S_Expression *s = new S_Expression;

  if (! depth) {
      maxdepth = 2 + (int) floor ((maxdepth-1) * random());
      if (strategy == RAMPED_HALF_AND_HALF) {
          if (random() < 0.5)   strategy = GROW;
          else                  strategy = FULL;
        }
    }
  ++depth;
  switch (strategy) {
      case GROW :
            if (depth == maxdepth)
                random_terminal (s);
            else random_terminal_or_function (s);
            break;
      case FULL :
            if (depth == maxdepth)
                  random_terminal (s);
            else random_function (s);
            break;
      case RAMPED_HALF_AND_HALF :
            cerr << "random_sexpression:  bad case\n";
            break;
    }
  if (s->type == STfunction)
     for (int i = 0;  i < Fset.nargs(s->which);  ++i)
        s->args[i] = random_sexpression (strategy, maxdepth,
                                         depth);
  return s;
}



// Chop off everything below a given depth
//
void S_Expression::restrict_depth (int maxdepth, int depth)
{
  ++depth;
  if (type == STfunction) {
    if (depth == maxdepth-1) {
       for (int i = 0;  i < Fset.nargs(which);  ++i)
         if (args[i]->type == STfunction) {
            delete args[i];
            args[i]=random_sexpression(GROW,maxdepth-depth);
          }
       }
    else
       for (int i = 0;  i < Fset.nargs(which);  ++i)
           args[i]->restrict_depth (maxdepth, depth);
}



// Put an ASCII representation of the S-expression into the
// given string
void S_Expression::write (char *s, int level)
{
  char buffer[64];
  if (! level)
      s[0] = 0;
  switch (type) {
      case STnone : cerr << "Error: bad case in
S_Expression::write\n";
                    break;
      case STconstant :   sprintf (buffer, "%g", val);
                          strcat (s, buffer);
                          break;
      case STterminal :   strcat (s, Tset.getname (which));
                          break;
      case STfunction :
            strcat (s, "(");
            strcat (s, Fset.getname (which));
            for (int i = 0;  i < Fset.nargs(which);  ++i) {
                strcat (s, " ");
                args[i]->write (s, level+1);
              }
            strcat (s, ")");
            break;
    }
}



// Stream output of S-expressions
ostream& operator<< (ostream& out, S_Expression *s)
{
  char buffer[10000];
  s->write (buffer);
  return (out << buffer);
}



////////////////////////////////////////////////////////
// The section below parses S-expressions
```

```
//////////////////////////////////////////////////////////

static int is_digit (char c)
{
  return ((c >= '0') && (c <= '9'));
}

static int is_alphanum (char c)
{
  return ((c >= 'a' && c <= 'z') ||
          (c >= 'A' && c <= 'Z') || is_digit (c));
}

static int is_white (char c)
{
  return (c == ' ' || c == '\n' || c == '\r' || c == '\t');
}

static int is_delimiter (char c)
{
  return (c == '('  ||  c == ')');
}



// Read a token from s, placing the token into *token.
// Return the type of token in *type (0 = end, '0' = number,
// 'a' = identifier, other character return as itself).
// Return the position where we stopped reading.
static char *get_token (char *token, char *s, char& type)
{
  char *tokenbegin = token;

  /* Skip whitespace */
  while (*s  &&  is_white (*s))
      s++;
  if (! *s)      // No more tokens
      type = 0;
  else if (is_digit (*s)  ||  *s == '.'  ||
           (*s == '-' && (is_digit (s[1]) || s[1] == '.')))
    {
      // It's a number (form: [-]d[d...][.[d...]] )
      type = '0';
      if (*s == '-')  *token++ = *s++;
      while (is_digit(*s))
          *token++ = *s++;
      if (*s == '.')  *token++ = *s++;
      while (is_digit(*s))
          *token++ = *s++;
    }
  else if (is_delimiter (*s)) {      // it's a delimeter
      type = *s;
      *token++ = *s++;
    }
  else {                 // It's an identifier or a function
      type = 'a';
      while (*s && !is_delimiter (*s) && ! is_white(*s))
          *token++ = *s++;
    }
  *token = '\0';
  return (s);
}



// Take a string and read an S-Expression from it (this is
// the private version used internally).
S_Expression *sexify (char **text)
{
  char   token[64];
  char   type;
  char  *s = get_token (token, *text, type);

  if (type == 0) {
      // No more tokens
      cerr << "sexify Error: Unexpected end of LISP
expression\n";
      return NULL;
    }
  S_Expression *se = new S_Expression;

  if (type == '0') {
      // It's a number
      se->type = STconstant;
      se->val = atof (token);
    }
  else if (type == 'a') {
      // It's a terminal
      se->type = STterminal;
      se->which = Tset.index (token);
      if (se->which  < 0) {
          cerr << "sexify Error: Unrecognized terminal " <<
token << '\n';
          delete se;
          return NULL;
        }
    }
  else if (*token == '(') {
      // It's a function
      s = get_token (token, s, type);
      se->type = STfunction;
      se->which = Fset.index (token);
      if (se->which < 0) {
          cerr << "sexify Error: Unrecognized terminal " <<
token << '\n';
```

```
          delete se;
          return NULL;
        }
      int nargs = Fset.nargs (se->which);
      for (int i = 0;  i < nargs;  ++i) {
        se->args[i] = sexify (&s);
        if (! se->args[i]) {
          cerr << "sexify Error: Bad function argument for "
               << token << '\n';
          delete se;
          return NULL;
         }
       }
      s = get_token (token, s, type);
      if (type != ')') {
          cerr << "sexify Error: Expected ')' not found\n";
          delete se;
          return NULL;
        }
    }
  else cout << "Huh?\n";
  *text = s;
  return (se);
}



// Take a string and read an S-Expression from it (this is
// the public version).
S_Expression *sexify (char *text)
{
  char *s = text;
  return sexify (&s);
}


EPHEMERAL ephemeral_constant = NULL;
```

4.5. gp.c --- Implementation of the GP Algorithm

```
////////////////////////////////////////////////////////
// gp.c - implementation for Genetic Programming in C++
// written by Larry I. Gritz, 1993
// The George Washington University, Dept. of EE&CS
// Computer Graphics and Animation Lab
////////////////////////////////////////////////////////
```

```
#include <iostream.h>
#include <math.h>
#include <stdio.h>
#include "gp.h"
#include "random.h"


float default_ephemeral_generator (void)
{
  // Return a number between -1.0 and 1.0
  return (2.0 * random() - 1.0);
}



// GP constructor
GP::GP (FITNESSFUNC fitfun, int pop_size)
{
  // Set up the default values
  M = pop_size;
  G = 51;
  pc = 0.9;
  pr = 0.1;
  pip = 0.9;
  Dcreated = 17;
  Dinitial = 6;
  pm = 0;
  pp = 0;
  fed = 0;
  pen = 0;
  dec_cond = NULL;
  pd = 0;
  generative_method = RAMPED_HALF_AND_HALF;
  reproduction_selection = FITNESS_PROPORTIONATE;
  second_parent_selection = FITNESS_PROPORTIONATE;
  use_greedy_overselection = (G >= 1000);
  overselection_boundary = ((G < 1000) ? 0.32 : (320 / M));
  use_elitist_strategy = 0;

  // Set up housekeeping info
  initialized = 0;
  pop = new Individual[M+1];
  newpop = new Individual[M+1];
  best_of_run.s = NULL;

  // Other defaults
  verbose = QUIET;
  termination_criteria = NULL;
  fitness_function = fitfun;
  standardize_fitness = NULL;
  sfit_dontreport = 1.0e20;
  generation_callback = NULL;
```

```
  bestworst_freq = 1;
  stat_filename = NULL;
  stat_file = NULL;
}



// GP Destructor
GP::~GP (void)
{
  if (stat_file)  fclose (stat_file);
  if (pop)        delete[M+1] pop;
  if (newpop)     delete[M+1] newpop;
}



// Initialize the population
void GP::init (void)
{
  if (stat_filename)
      stat_file = fopen (stat_filename, "wt");
  float ptotal = pr + pc + pm + pp + pen;
  if (ptotal < 0.001)
      cout << "Error: (pr + pc + pm + pp + pen) should sum to
1.0\n";
  pr /= ptotal;
  pc /= ptotal;
  pm /= ptotal;
  pp /= ptotal;
  pen /= ptotal;
  best_of_run.sfit = 1.0e20;
  gen = 0;
}



// Create the generation 0 population
void GP::create_population (void)
{
  if (verbose & TELL_INITIALIZE)
      cout << "Creating initial population...\n";
  for (int i = 0;  i < M;  ++i) {
      pop[i].s = random_sexpression (generative_method,
                                     Dinitial, 0);
      // Make sure we don't have a duplicate
      int duplicated = 0;
      for (int j = 0;  j < i && !duplicated;  ++j)
          if (equiv (pop[i].s, pop[j].s)) {
              delete pop[i].s;
              pop[i].s = NULL;
              duplicated = 1;
            }
      if (duplicated) {
          --i;
          continue;
        }
      pop[i].recalc_needed = 1;
      newpop[i].s = NULL;
      if (verbose & LIST_INITIAL_EXPRESSIONS) {
          cout << i << ": " << pop[i].s << '\n';
          cout.flush();
        }
    }
}



void GP::list_parameters (void)
{
  if (verbose & LIST_PARAMETERS) {
      cout << "Parameters used for this run:\n";
      cout << "=============================\n";
      cout << "Max generations: " << G << '\n';
      cout << "Population size: " << M << '\n';
      cout << "Max depth for new individuals: "
           << Dinitial << '\n';
      cout << "Max depth for crossed and mutated individuals: "
           << Dcreated << '\n';
      cout << "Fitness-proportionate reproduction fraction: " <<
pr << '\n';
      cout << "Crossover fraction: " << pc << " ("
           << pip*10.0 << "% at internal points)\n";
      cout << "Mutation fraction: " << pm << '\n';
      cout << "Permutation fraction: " << pp << '\n';
      cout << "Encapsulation fraction: " << pen << '\n';
      cout << "Selection method: ";
      switch (reproduction_selection) {
          case UNIFORM: cout << "uniform\n"; break;
          case FITNESS_PROPORTIONATE:
                 cout << "fitness-proportionate\n"; break;
          case TOURNAMENT: cout << "tournament, size = "
                                << tournament_size << '\n';
                           break;
          case RANK: cout << "rank\n"; break;
        }
      cout << "Generation method: ";
      switch (generative_method) {
          case GROW: cout << "grow\n"; break;
          case FULL: cout << "full\n"; break;
          case RAMPED_HALF_AND_HALF:
                cout << "ramped half-and-half\n"; break;
        }
    }
}
```

```
// Given a particular selection method, choose a random
// member of the population and return its index.
static int choose_random (GP *gp, SelectionMethod method)
{
  int which = -1;
  int i, j;

  switch (method) {
      case FITNESS_PROPORTIONATE :
          float f = random();
          if (gp->use_greedy_overselection) {
               // 80% of the time, select from best
              if (random() < 0.8)
                  f *= gp->overselection_boundary;
                else  // 20% of the time, use the rest
                  f = gp->overselection_boundary +
                      f * (1.0 - gp-overselection_boundary);
            }
          for (j = 0;  j < gp->M;  ++j)
              if (f <= gp->pop[j].sumnfit)
                  return j;
          cerr << "Ran past end in choose_random "<<f<<"\n";
          cerr.flush();
          // Purposely go to next case if we hit this
      case UNIFORM :
          which = (int) floor (random() * gp->M);
          break;
      case TOURNAMENT :
          for (i = 0;  i < gp->tournament_size;  ++i) {
            j = (int) floor (gp->M * random());
            if (!i || gp->pop[j].nfit > gp-?pop[which].nfit)
               which = j;
           }
          break;
    }
  if (which < 0 || which >= gp->M) {
      cerr << "Ug!  invalid choice in choose_random\n";
      return 0;
    }
  return which;
}



// Sort pop by normalized fitness value.  Do a selection
// sort, using newpop as temporary storage.
void GP::sort_fitness (void)
{
  float total = 0;

  for (int i = 0;  i < M;  ++i) {
      // Invariant: pop[0..i-1] contains the i largest
      //fitnesses, sorted
      int biggest = i;
      for (int j = i+1;  j < M;  ++j)
          if (pop[j].nfit > pop[biggest].nfit)
              biggest = j;
      // Now j contains the biggest element in [i+1..M-1] so
      // we swap elem i with biggest element in [i+1..M-1].
      if (biggest != i) {
          S_Expression *temp1 = pop[i].s;
          S_Expression *temp2 = pop[biggest].s;
          pop[i].s = pop[biggest].s = NULL;
          Individual temp = pop[i];
          pop[i] = pop[biggest];
          pop[biggest] = temp;
          pop[i].s = temp2;
          pop[biggest].s = temp1;
        }
      total += pop[i].nfit;
      pop[i].sumnfit = total;
      // Now pop[0..i] contains i+1 largest fitnesses
    }
}



// Create the next generation of individuals
void GP::nextgen (void)
{
  S_Expression *s, **parentptr;

  if (use_greedy_overselection)
      sort_fitness ();
  for (int i = 0;  i < M;  ++i) {
      if (! i  &&  use_elitist_strategy) {
          newpop[i] = best_of_run;
          continue;
        }

      // All ops require reproducing one individual first
      newpop[i] = pop[choose_random (this,
                                     reproduction_selection)];
      float option = random();
      if (option <= pc) {
          // Crossover operation
          newpop[i+1] = pop[choose_random (this,
                                           second_parent_selection)];
          crossover (&(newpop[i].s), &(newpop[i+1].s), pip);
          newpop[i].s->restrict_depth (Dcreated);
          newpop[i].recalc_needed = 1;
          newpop[i+1].s->restrict_depth (Dcreated);
          newpop[i+1].recalc_needed = 1;
```

```
        ++i;
      }
    // Any non-plain varieties of reproduction?
    else if (option <= (pc+pm)) {
        // Mutation operation
        int depth, total, internal, external, n = -1, m;
        newpop[i].s->characterize (&depth, &total,
                                   &internal, &external);
        m = (int) floor (total * random());
        s = newpop[i].s->selectany (m, &n, &parentptr);
        if (! parentptr)
            parentptr = &(newpop[i].s);
        *parentptr = random_sexpression (GROW, 6);
        delete s;
      }
    else if (option <= (pc+pm+pp)) {
        // Permutation operation
        s = newpop[i].s->select (1.0, &parentptr);
        s->permute();
      }
    else if (option <= (pc+pm+pp+pen)) {
        // Encapsulation operation
        // For now, just do reproduction
        s = newpop[i].s->select (1.0, &parentptr);
        if (! parentptr)
            parentptr = &(newpop[i].s);
        int e = Fset.encapsulate (s);
        delete s;
        s = new S_Expression;
        s->type = STfunction;
        s->which = e;
        *parentptr = s;
      }
    // else Just plain reproduction, don't do any additional
work
   }
  // Swap the current and next generations
  Individual *temp = pop;
  pop = newpop;
  newpop = temp;
}



// Evaluate the fitness of each individual in the population
void GP::eval_fitnesses (void)
{
  float total_afitness = 0.0;
  int i;

  bestofgen_sfit = 1.0e20;
  worstofgen_sfit = -1.0e20;
  avgofgen_sfit = 0;
  for (i = 0;  i < M;  ++i) {
      if (pop[i].recalc_needed) {
          pop[i].rfit = (*fitness_function)(pop[i].s,
                         &(pop[i].hits));
          if (standardize_fitness)
              pop[i].sfit =
                  standardize_fitness (pop[i].rfit);
          else pop[i].sfit = pop[i].rfit;
          pop[i].afit = 1.0 / (1.0 + pop[i].sfit);
          pop[i].recalc_needed = 0;
        }
      total_afitness += pop[i].afit;
      avgofgen_sfit += pop[i].sfit;
      if (pop[i].sfit < bestofgen_sfit) {
          bestofgen_sfit = pop[i].sfit;
          bestofgen_index = i;
          bestofgen_hits = pop[i].hits;
        }
      else if (pop[i].sfit > worstofgen_sfit)
          worstofgen_sfit = pop[i].sfit;
      if (verbose & LIST_GENERATIONAL_FITNESSES  &&
          pop[i].sfit < sfit_dontreport) {
          cout << "\t\t" << i << ": ";
          cout << pop[i].rfit << " " << pop[i].afit << "  "
               << pop[i].hits << " hits";
          if (bestofgen_index == i)
              cout << "  (best-of-generation)";
          if (pop[i].sfit < best_of_run.sfit)
              cout << "  (best-so-far)";
          cout << '\n';
          cout.flush();
        }
    }
  avgofgen_sfit /= (float)M;
  float total = 0;
  for (i = 0;  i < M;  ++i) {
      pop[i].nfit = pop[i].afit / total_afitness;
      total += pop[i].nfit;
      pop[i].sumnfit = total;
    }

  if (bestofgen_sfit < best_of_run.sfit) {
      best_of_run = pop[bestofgen_index];
      bestofrun_gen = gen;
    }
  if (stat_file)
      fprintf (stat_file, "%d %g %g %g\n", gen,
         bestofgen_sfit, worstofgen_sfit, avgofgen_sfit);
}
```

```
void GP::report_on_run (void)
{
  if (verbose & END_REPORT) {
      cout << "\nThe best-of-run individual program was”
          << “ found on generation " << bestofrun_gen
          << "\nand had a standardized fitness measure of "
         << best_of_run.sfit << " and " << best_of_run.hits
          << " hits:\n";
      cout << best_of_run.s << '\n';
      if (verbose & SHOW_EDITED_BEST) {
          cout << "The edited version of the best-of-run
individual is:\n";
          S_Expression *edited = best_of_run.s->copy();
          edited = edit (edited);
          cout << edited << "\n\n";
          delete edited;
        }
      cout.flush();
    }
}



// The actual GP run is invoked with "go":
void GP::go (int maxgens)
{
  char buffer[4000];

  G = maxgens;
  if (gen == 0) {
      init ();
      create_population ();
    }

  for ( ;  gen <= G;  ++gen) {
      int report;  // 1 if we report info on this generation
      if (bestworst_freq == 0)
          report = 0;
      else if (bestworst_freq == 1)
          report = 1;
      else report = !(gen % bestworst_freq);
      if (gen > 0)
          nextgen ();
      if (verbose & GENERATION_UPDATE) {
          cout << "\rGeneration " << gen << ' ';
          cout.flush();
        }
      eval_fitnesses();
      if (report  &&  (verbose & GENERATION_UPDATE)) {
          cout << "\n--------------\n";
          cout << "average standardized fitness of gen was "
               << avgofgen_sfit << ".\n";
          cout << "worst of gen had standardized fitness "
               << worstofgen_sfit << ".\n";
          cout << "best of gen had standardized fitness "
               << bestofgen_sfit << " and "
                << bestofgen_hits << " hits:\n";
          pop[bestofgen_index].s->write (buffer);
          cout << buffer << "\n";
          if (verbose & SHOW_EDITED_BEST) {
             S_Expression *s=pop[bestofgen_index].s->copy();
             s = edit (s);
             s->write (buffer);
             cout << "\nEdited = " << buffer << "\n";
             delete s;
            }
          cout << '\n';
        }
      else if (verbose & GENERATION_UPDATE) {
          cout << "(best sfit " << best_of_run.sfit << ", "
               << best_of_run.hits << " hits)       ";
        }
      cout.flush();
      if (generation_callback)
          generation_callback (this);
      if (termination_criteria)
          if ((*termination_criteria)(this))
              break;
    }
  if (verbose & GENERATION_UPDATE) {
      cout << '\n';
      cout.flush();
    }
  report_on_run();
}



// Debugging tool: print the entire population and fitness
void GP::print_population (void)
{
  for (int i = 0;  i < M;  ++i)
      cout << i << ": [sfit " << pop[i].sfit << "] "
           << pop[i].s << '\n';
}
```

4.6. stdfunc.c --- Canned functions

```
////////////////////////////////////////////////////////
// stdfunc.c -- implementations of standard functions for GP
//
```

```
// Note that the editing operation is implemented for a few
// functions as examples.  The editing implementations of
// the remaining functions are not shown.
//
// written by Larry I. Gritz, 1993
// The George Washington University, Dept. of EE&CS
// Computer Graphics and Animation Lab
////////////////////////////////////////////////////////////

#include <iostream.h>
#include <malloc.h>
#include <string.h>
#include <math.h>
#include <stdio.h>
#include "gp.h"


// Macro to return a floating point value and kill the rest
// of the subtree
inline S_Expression *retnum (float f, S_Expression *s)
{
  S_Expression *s2 = new S_Expression;
  s2->type = STconstant;
  s2->val = f;
  delete s;
  return s2;
}


// Macro to return one arg and kill the rest of the subtree
inline S_Expression *retarg (int a, S_Expression *s)
{
  S_Expression *s2 = s->args[(a)];
  s->args[(a)] = NULL;
  delete s;
  return s2;
}



////////////////////////////////////////////////////////////
//  Addition function
////////////////////////////////////////////////////////////

static float plus (S_Expression **args) {
  return (args[0]->eval() + args[1]->eval());
}

static S_Expression *edit_plus (S_Expression *s)
{
  float f, f2;
  // Rule 1: if either arg is 0, result is other arg
  if (s->args[0]->is_numerical (f)  &&  f == 0.0)
      return retarg (1, s);
  if (s->args[1]->is_numerical (f)  &&  f == 0.0)
      return retarg (0, s);
  // Rule 2: if both args are numbers, result is their sum
  if (s->args[0]->is_numerical(f) &&
         s->args[1]->is_numerical (f2))
      return retnum (f+f2, s);
  return s;
}

void use_addition (void) {
  Fset.add ("+", 2, plus, edit_plus);
}



////////////////////////////////////////////////////////////
//  Subtraction function
////////////////////////////////////////////////////////////

static float minus (S_Expression **args) {
  return (args[0]->eval() - args[1]->eval());
}

static S_Expression *edit_minus (S_Expression *s)
{
  float f, f2;

  // Rule 1: if second arg is 0, result is first argument
  if (s->args[1]->is_numerical (f)  &&  f == 0.0)
      return retarg (0, s);
  // Rule 2: if both args are numbers, result is a number
  if (s->args[0]->is_numerical (f) &&
          s->args[1]->is_numerical (f2))
      return retnum (f-f2, s);
  // Rule 3: if both args are the same identifier result=0
  if (s->args[0]->type == s->args[1]->type  &&
      s->args[0]->type == STterminal &&
      s->args[0]->which == s->args[1]->which)
          return retnum (0, s);
  return s;
}

void use_subtraction (void) {
  Fset.add ("-", 2, minus, edit_minus);
}



////////////////////////////////////////////////////////////
//  Multiplication function
////////////////////////////////////////////////////////////
```

```
static float mult (S_Expression **args) {
  return (args[0]->eval() * args[1]->eval());
}

static S_Expression *edit_mult (S_Expression *s)
{
  float f, f2;

  /* Rule 1:  if either argument is 0, result is 0 */
  if ((s->args[0]->is_numerical (f)  &&  f == 0.0)  ||
      (s->args[1]->is_numerical (f2) && f2 == 0.0))
          return retnum (0, s);
  // Rule 2: if both args are numbers, result is a number
  if (s->args[0]->is_numerical (f)  &&
          s->args[1]->is_numerical (f2))
      return retnum (f*f2, s);
  // Rule 3: if either arg is 1, result is other argument
  if (s->args[0]->is_numerical (f)  &&  f == 1.0)
      return retarg (1, s);
  if (s->args[1]->is_numerical (f)  &&  f == 1.0)
      return retarg (0, s);
  return s;
}

void use_multiplication (void) {
  Fset.add ("*", 2, mult, edit_mult);
}



////////////////////////////////////////////////////////
//  Protected division function
////////////////////////////////////////////////////////

static float protected_div (S_Expression **args)
{
  float denom = args[1]->eval();
  if (denom == 0.0)
       return 1;
  else return (args[0]->eval() / denom);
}

S_Expression *edit_protected_division (S_Expression *s)
{
  float f, f2;

  // Rule 1: if second argument is 0, result is 1
  if (s->args[1]->is_numerical (f)  &&  f == 0.0)
      return retnum (1, s);
  // Rule 2:  if first argument is 0, result is 0
  if (s->args[0]->is_numerical (f)  &&  f == 0.0  &&
      ! s->args[1]->side_effects())
          return retnum (0, s);
  // Rule 3: if both args are non-zero numbers, result is a
number
  if (s->args[0]->is_numerical (f)  &&
          s->args[1]->is_numerical (f2))
      return retnum (f/f2, s);
  // Rule 4: if second arg is 1, result is first argument
  if (s->args[1]->is_numerical (f)  &&  f == 1.0)
      return retarg (0, s);
  // Rule 5: if both args are the same terminal, result is 1
  if (s->args[0]->type == s->args[1]->type  &&
      s->args[0]->type == STterminal &&
      s->args[0]->which == s->args[1]->which)
          return retarg (1, s);
  return s;
}

void use_protected_division (void) {
  Fset.add ("%", 2, protected_div, edit_protected_division);
}



//  Sine function
static float sine (S_Expression **args) {
  return sin (args[0]->eval());
}

void use_sin (void) { Fset.add ("sin", 1, sine); }



//  Cosine function
static float cosine (S_Expression **args) {
  return cos (args[0]->eval());
}

void use_cos (void) { Fset.add ("cos", 1, cosine); }



//  ATG -- protected arctangent with 2 arguments
static float atg (S_Expression **args) {
  return atan2 (args[0]->eval(), args[1]->eval());
}

void use_atg (void) { Fset.add ("atg", 2, atg); }



//  Exponential function
static float expfun (S_Expression **args) {
  float f = args[0]->eval();
```

```
  return exp (f);
}

void use_exp (void) { Fset.add ("exp", 1, expfun); }



//  Protected logarithm function
static float rlog (S_Expression **args) {
  float x = args[0]->eval();
  return (x > 0.0 ? log (x) : 0);
}

void use_rlog (void) { Fset.add ("rlog", 1, rlog); }



// absolute value function
static float abs_imp (S_Expression **args) {
  return fabs (args[0]->eval());
}

void use_abs (void) { Fset.add ("abs", 1, abs_imp); }



// greater-than -- return 1 if first arg is > second arg,
//   else return -1
static float gt (S_Expression **args) {
  return (args[0]->eval() > args[1]->eval()) ? 1 : -1;
}

void use_gt (void) { Fset.add ("gt", 2, gt); }



// Logical AND function
static float logical_and (S_Expression **args)
{
  if (! args[0]->eval()) return 0;
  else return (args[1]->eval());
}

void use_and (void) { Fset.add ("and", 2, logical_and); }



// Logical OR function
static float logical_or (S_Expression **args)
{
  if (args[0]->eval()) return 1;
  else return (args[1]->eval());
}

void use_or (void) { Fset.add ("or", 2, logical_or); }



// Logical NOT function
static float logical_not (S_Expression **args) {
  return (! args[0]->eval());
}

void use_not (void) { Fset.add ("not", 1, logical_not); }



// Logical IF function
static float logical_if (S_Expression **args)
{
  if (args[0]->eval() != 0)
       return (args[1]->eval());
  else return (args[2]->eval());
}

void use_if (void) { Fset.add ("if", 3, logical_if); }



// Logical EQ -- equality test
static float logical_eq (S_Expression **args)
{
  if (args[0]->eval() == args[1]->eval())
       return 1;
  else return 0;
}

void use_eq (void) { Fset.add ("eq", 2, logical_eq); }



// PROGN2 -- sequence of 2 moves
// PROGN3 -- sequence of 3 moves

static float progn2 (S_Expression **args)
{
  args[0]->eval();
  return args[1]->eval();
}

void use_progn2 (void)
{  Fset.add ("progn2", 2, progn2);  }

static float progn3 (S_Expression **args)
{
  args[0]->eval();
```

```
  args[1]->eval();
  return args[2]->eval();
}

void use_progn3 (void)
{  Fset.add ("progn3", 3, progn3);  }


// SRT protected square root
static float srt (S_Expression **args) {
  return sqrt (fabs (args[0]->eval()));
}

void use_srt (void) {  Fset.add ("srt", 1, srt);  }



// SQ square operator

static float sq (S_Expression **args)
{
  float f = args[0]->eval();
  return f*f;
}

void use_sq (void) {  Fset.add ("sq", 1, sq);  }



// CUB cube operator
static float cub (S_Expression **args)
{
  float f = args[0]->eval();
  return f*f*f;
}

void use_cub (void) {  Fset.add ("cub", 1, cub);  }



// SIG sign operator
static float sig (S_Expression **args) {
  return ((args[0]->eval() > 0.0) ? 1.0 : -1.0);
}

void use_sig (void) {  Fset.add ("sig", 1, sig);  }



// IFLTZ -- if less than zero
 static float ifltz (S_Expression **args)
{
  if (args[0]->eval() < 0)
      return (args[1]->eval());
  else return (args[2]->eval());
}

void use_ifltz (void) { Fset.add ("ifltz", 3, ifltz); }



// IFLTE -- if less than or equal
static float iflte (S_Expression **args)
{
  if (args[0]->eval() <= args[1]->eval())
      return (args[2]->eval());
  else return (args[3]->eval());
}

void use_iflte (void) { Fset.add ("iflte", 4, iflte); }



// SREXPT -- raise to powers
static float srexpt (S_Expression **args)
{
  float f = args[0]->eval();
  if (fabs (f) > 1.0e-9)
      return pow (f, args[1]->eval());
  else return 0.0;
}

void use_srexpt (void) { Fset.add ("srexpt", 2, srexpt); }



// DU -- looping operator
int max_du_iterations = 25;

static float du (S_Expression **args)
{
  for (int iters = 0;  args[1]->eval();  ++iters) {
      args[0]->eval();
      if (iters >= max_du_iterations)
          return 0;
    }
  return 1;
}

void use_du (void) { Fset.add ("du", 2, du); }
```

5. Bibliography

[1] Darwin, Charles. The Origin of Species. Mentor, 1859.

[2] Davidor, Yuval. *Genetic Algorithms and Robotics: A Heuristic Strategy for Optimization*. World Scientific, 1991.

[3] Davis, Lawrence, ed. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[4] Dawkins, Richard. *The Blind Watchmaker*. Harlow Logman, 1986.

[5] Dawkins, Richard. The evolution of evolvability. In *Artificial Life Proceedings*, pages 201-220, 1987.

[6] Dawkins, Richard. *The Selfish Gene*. Oxford University Press, 1976.

[7] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[8] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[9] Koza, John R. *Genetic Programming*. MIT Press, 1992.

[10] Michalewicz Zbigniew. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1992.

[11] Steele, G. *Common Lisp, The Language*. Digital Press, 1984.

[12] Stroustrup, Bjarne. *The C++ Programming Language.* Addison-Wesley, 2nd ed., 1991.