Object oriented techniques in genetic algorithms for optimization

Kevin S. Lohn
The University of Montana

Object Oriented Techniques in Genetic Algorithms for Optimization

by

Kevin S. Lohn

B.S., University of Montana, 1983

Presented in partial fulfillment of the requirements

for the degree of

Master of Science

in Computer Science

University of Montana

1991

Approved by

[illegible]

Chairman, Thesis Committee

[illegible]

Dean, Graduate School

Feb. 27, 1991

Date

Lohn, Kevin S., MS, March 1991 Computer Science

Object Oriented Techniques in Genetic Algorithms for Optimization (102 pp.)

Director: Alden H. Wright [illegible]

This paper concerns the invention of several object oriented structures and their application to simulation programs; specifically genetic algorithms. The application of these constructs then contribute to a working program and an examination of two implementations of genetic algorithms: Binary Genetic Algorithms and Real Vector Genetic Algorithms. The examination shows that it is possible to construct a model of a binary encoded genetic algorithm using only vectors of floating point numbers.

The need for a reporting and monitoring system in a simulation prompts the development of Probes. These objects consist of associations of code and data that can be dynamically attached to the program at run time. This means that reporting facilities gain significant flexibility.

The users' interaction in controlling experiments in a simulation motivate the invention of Iterators. Iterators serve as an intermediary between the user and the simulation. They provide the user with a consistent interface, while providing the simulation with a uniform control structure.

The structure of the genetic algorithm program demonstrates the power of an object oriented framework. The details of the construction of the framework are discussed along with techniques for extending them.

The program is set up for the implementation of two genetic algorithm variants. These variants are then compared using several of the classic deJong problems.

Table Of Contents

1. Introduction

Programs from the class of programs known as simulations share some common features. They require a reporting and monitoring system that facilitates rapid incremental program development and debugging, a user interface that allows flexible control of experiments, and a flexible overall framework that allows easy program maintenance and modification. There has been little effort to create standardized techniques to address these needs. Object oriented programming languages provide a powerful base on which to build these tools. This paper explores several object oriented constructs that can serve to meet the requirements of simulation programs. By applying these constructs to an optimization program using a genetic algorithm, their power is demonstrated.

The first constructs discussed in this paper are Probes. Probes are defined as a uniform method of implementing statistical measurements, reports and debugging code. Consisting of associations of executable code and data, they can be dynamically hooked to locations in the program at run time. They perform their functions whenever the code to which they are attached is active.

Iterators are defined and implemented as intermediaries between the needs of the user and the requirements of the simulation. They provide a uniform interface that allows the user to control experiments with the simulation. In the program they provide a control structure for experiments, isolating them from the user.

The third section discusses the framework of the genetic algorithm program. Extensive use of the object oriented concepts of encapsulation and inheritance as well as the exploitation of Iterators and Probes makes the program extremely flexible. Implementation of a wide variety of genetic algorithm variants is facilitated by the reusable code concept. Several inheritance hierarchies are implemented within the program.

Once the program is completely defined, an experiment is conducted by implementing Real Vector and Binary Genetic Algorithms within the program framework. The goal is to construct a Real Vector Genetic Algorithm that closely simulates a Binary Genetic Algorithm. The experiment consists of using each of the two techniques to optimize several of the classic deJong test problems. The performance of each of the two techniques is then compared. Success is measured by how closely the Real Vector Genetic Algorithm mimics the Binary Genetic Algorithm.

1.1. Object Oriented Programming Background

Object oriented programming (OOP) is a programming paradigm that is data driven. Rather than constructing a hierarchy of functions and procedures that invoke each other, an OOP program establishes a hierarchy of data structures and their interactions. OOP languages enable a closer link between the real world problem and the program.

The objects of object oriented programming are a metaphor for objects in the real world. They are organized into classes of objects of the same type. A class is essentially a type that defines a data structure for the properties of a real world object. For example, if a model were to be constructed of a railroad, one class might be a GenericRailroadCar. The properties, called instance variables, could be data such as Owner, NumberOfWheels and Length. It is then possible to create objects of the type GenericRailroadCar. An object of a type that is a class is called an instance of the class. Instances can be manipulated like any other data structure.

One important feature of OOP languages, is the ability to define the interface of an object. The interface is a set of functions (methods, in some languages) that manipulate the data of an object. Generally, these functions have exclusive access to the data of the object. If any other function needs to access the data of an object, it must do so through the functions that define the object's interface. No outside function needs to know about the internal representation of the data within a class. A change can be made in the internal representation without changing the interface. For example, an internal linked list could be changed to a tree structure transparently. This facilitates incremental development and simplifies program maintenance.

Inheritance is another important feature of OOP languages. In the railroad car example above, a very generic rail car was defined. A box car definition should contain the generic rail car definition in addition to several other features. CubicCapacity and NumberOfDoors might be appropriate. A tank car definition may need FluidCapacity and MaximumPressure. Both TankCar and BoxCar can be made to "inherit" the properties of a generic railroad car. Each of the two new classes also have their own interfaces to their properties. The generic railroad car class is referred to as the base class. BoxCar and TankCar are derived classes. The depth of inheritance can be greater than two levels; it would be simple to create a specialized type of TankCar with additional properties. It is also possible to inherit from more than one class. It would be helpful for TankCar and BoxCar to inherit from a linked list node class. This would enable trains to be linked together from many types of railroad cars. The facility of inheritance encourages reusability of code and rapid prototyping.

Derived classes can redefine functions of the interface. The redefinitions supersede those defined in the base class. For example, the GenericRailroadCar class might have a function called IsHazardous that always returns the value "no". By default, all classes derived from GenericRailroadCar will respond "no" to the IsHazardous function. The TankCar class might redefine the function to check the contents of the tank car and compare it with a list of known hazardous materials. An instance of TankCar would not respond "no" by default, it would respond as determined by its redefinition of the IsHazardous function.

Consider a train composed of a linked list of railroad cars. To determine if a train carries hazardous materials it must apply the IsHazardous function to all of its railroad cars. The train does not have to know the types of cars it contains, it must just traverse its list applying the IsHazardous function to all cars. The object oriented language takes care of the details of calling the correct version of the IsHazardous function for each car. A TankCar instance will use the TankCar version of the IsHazardous function. Railroad cars that do not have a redefinition of the IsHazardous function, will use the default function from the GenericRailroadCar class. This feature is called virtual functions in C++.

2. Measuring and Reporting - Probes

A program implementing a genetic algorithm takes initial data and transforms it to a new form. The transformation may take many hours and consist of millions of individual steps. Because the algorithm produces no output until the very end, it is difficult to know how the algorithm is progressing. Both the processes of debugging the program and experimenting with the program can suffer from a lack of methods of measuring and reporting on the algorithm's progress.

The process of conceiving and implementing reports regarding the progress of a program through an algorithm, is both time consuming and difficult. Each report requires its own unique supporting structure. As more reports are implemented, the original code begins to suffer from excessive modification. The frequent changes can breed elusive bugs. It is apparent that an important priority is to create a generalized method for the creation of new performance measures that is flexible, minimizes impact on existing code and is easy to manipulate.

2.1. Stages of Reporting

The processes of generating reports have common elements regardless of content of the report. Reporting consists of four stages: initialization in preparation of gathering data, gathering data from the source, manipulating the data and generating a report. As an example, consider the generation of an average temperature report for a weather monitoring station.

The first step is used to reset the statistical measures that makeup the report. To calculate an average temperature, two variables are needed: the sum of the temperatures and a counter for the number of temperatures in the sum. This first step would set both values to zero.

The second step, consisting of gathering data, is typically repeated many times. The data is either stored or processed immediately. For example, weather monitoring stations collect data about current conditions many times per day. During this step for an average temperature report, the current temperature is added to the running temperature total and the counter is incremented.

The processing is the third of the four stages. It transforms the data into a more useful form. This step can be coupled directly with the second stage or postponed until it can process data in a batch. It is even possible to combine this stage with the final reporting stage. For a weather monitoring station, this third stage could consist of actually making the calculation of an average wind speed or temperature.

The final stage is the actual report generation. This can consist of the dissemination of either the raw data or the data as it was transformed by the third stage. Generally, the report is routed to a file or printer, but there is no reason that the output cannot be routed to another process.

2.2. Implementation Options

There are several options in implementing the four stages of report generation. All of the methods, however, have an intrusive nature in common. The object that is the source of the data cannot be monitored entirely passively without some sort of concurrent processing ability. Most conventional computer languages do not have this inherent ability. It is necessary for the monitoring target to cooperate in the monitoring process. This necessarily consists of modifying the code associated with the target .

2.2.1. Global Method

The least desirable method of implementing a report involves the extensive use of global variables. If the goal is to calculate the average of a value over several iterations of a loop in an existing program, a global variable can serve as a sum and another can serve as an iteration counter. The code to initialize the variables, collect the values, calculate the averages and then generate the report must be embedded into the existing program.

This method obviously has several drawbacks. It requires that the existing code be modified and recompiled. If the source code is unavailable for modification, this technique cannot work. If the source code is available, its modification in implementing a statistical measure can introduce side effects. If the statistical measure is temporary or needed only periodically, it is undesirable to clutter the original code. Clutter makes the original code difficult to maintain, especially if there are several of these statistical measures implemented.

In conventional languages (not object oriented) the data from which a report is generated is not necessarily found grouped together. The code required to collect data may have to be scattered throughout a wide set of routines. It is very tempting to embed the data collection code in the routine that is the most convenient. While that choice might make the report easier to create, it is not necessarily the best choice when attempting to maintain a program's strict modularity.

The program controlling a weather station might consist of one large loop in the main line code. Inside the loop, routines are called to control the various devices in the weather station. The thermometer, the barometer and the wind gauge might each have controlling routines that are called repeatedly in the loop. Figure 2.1 gives an example.

```
main loop
call the thermometer routine
call the barometer routine
call the wind gauge routine
end main loop
```

FIGURE 2.1

The addition of just three reports can complicate the code to the point where the original purpose is lost in the clutter. Here, in Figure 2.2, daily average temperature, monthly average temperature and daily average wind speed reports are inserted.

```
main loop
  if it is a new day
    reset daily average temperature variables
    reset average wind speed variables
  end if
  if it is a new month
    reset monthly average temperature variables
  end if
  call the thermometer routine
  call the barometer routine
  call the wind gauge routine
  if it is the end of a day
    calculate the daily average temperature
    report the daily average temperature
    calculate the daily wind speed
    report the daily wind speed
  end if
  if it is the end of the month
    calculate the monthly average temperature
    report the monthly average temperature
  end if
end main loop
```

FIGURE 2.2

The reports must execute their first stage (reset) functions inside the loop. But because the routines should not be called on every iteration, conditionals are added to restrict their execution. The data collection routine will have to be placed inside the appropriate routine for the gauge being monitored. The temperature averaging routines need to collect data inside the thermometer routine. The average wind speed report must collect its data from the wind gauge routine. The calculation and reporting routines must also be placed inside the loop. Again, their execution must be restricted by conditionals.

This means that the variables used to calculate an average must be accessible not only from the main loop, but from the appropriate gauge's routine. Hence the necessity for global definitions of these variables. As the number of reports multiples, the task of administering the global variables expands to become unmanageable.

2.2.2. OOP Method One

Object oriented programming allows for the encapsulation of related data. The running total and iteration counter of a measurement calculating an average can be set up to be visible only to a certain group of routines. These variables, called instance variables, are local to an object. Only the routines that are designed for the object can have access to its instance variables.

The weather monitoring station example could be an object oriented program . One class could represent the thermometer, another class could represent the wind gauge. Average temperature is straightforward to implement. The sum and counter variables can be added as an instance variables to the thermometer class. When the thermometer's code is running, the code to calculate the average will also run. Of course, it is still necessary to restrict the average calculation with conditionals so that it will be calculated only at appropriate times.

Using instance variables to implement statistical measures and reports only partially solves the problems of the global implementation. The intrusive code is internal to one class. While this encapsulation may aide in avoiding undesirable side effects, it still requires that the source code for the class be available. The routines to collect data will still need to be embedded into the original code. In addition, the reduction of the cluttering comes at the expense of diminished power over the global technique.

The possibility that a report may need to gather information from more than one class complicates this technique. New methods must be added to all classes involved in the statistical measurement to enable the data to be collected. Ambiguities also arise regarding the proper location of the instance variables. Reasons for selecting one class over another for the location of the instance variables become complex.

The calculation of wind chill is a report that is not easy to implement. If an instance variable for wind chill is set up in the temperature class, a method of retrieving the wind speed from the wind gauge class needs to be devised. Adding this method seems to violate the basic modularization that object oriented programming provides. The temperature class should consist of methods for monitoring and reporting temperatures. The only reason to add a method for retrieving the wind speed, is for a report that only incidentally needs the current temperature. The same problem occurs if the instance variable is moved to wind gauge class.

2.2.3. OOP Method Two

A step in the right direction is to make a class for the statistical measurement itself. It can encapsulate all of the variables necessary for the calculation of the statistic in one place. The routines for actually calculating the average are then also encapsulated in the class.

In the weather station example, a wind chill class can be created. When it needs to make its calculation of wind chill, it simply queries the temperature class and the wind gauge class for their current values. No routines need to be added to either of those classes, therefore their source code is not needed. This technique reduces the number of locations that the intrusive code might be found. The only intrusive aspect of this implementation of a wind chill report, is adding the code to activate the appropriate methods of the wind chill class at the proper times.

Cluttering is only slightly better with this technique than with the global method. Since each statistical measure must go through the four stages of the reporting process, there will exist at least four lines of code added to the source code somewhere. As the number of statistical measures increases, the intrusive code problem can cause difficulties. Again, with temporary measures or measures needed only periodically, the source code becomes unreadable.

The intrusive code consists of a reference to a measurement and an associated function for that measure. An example of a measurement would be the class that calculates WindChill. An associated function would be its routine that queries the thermometer and the wind gauge. Think of these as a measurement/function pair. The WindChill class and its routine that generates the printed report would be another measurement/function pair. The function part of a measurement/function pair is directly related to one of the four stages of reporting.

This technique does not address the problem of monitoring an object while it is performing one of its own tasks. A measurement/function pair could be inserted into the code of an object, but the code will still have to be modified every time a new measurement or report is needed. While this modification might be simpler than with the global technique, it still has many of the same problems.

The goal is to create a technique that will treat all measurements and reports the same way. Modifying the code of an existing object should be limited to just once. When a new report is conceived, existing objects should be capable of exploiting it without modification.

2.2.4. OOP Method Three

The solution is to treat measurement/function pairs as data. Object oriented programming facilitates a technique of dynamically creating and manipulating an association of an object and one of its functions. Once defined, these associations can be treated like any other data. It is possible to create arrays or linked lists of them. At any time, the function in the association can be invoked in a generic way.

The pseudo code example of figure 2.3 is designed to demonstrate the flexibility of the measurement/function pair concept. In the figure, a variable called MFP of the type MeasurementFunctionPair is created. This variable will hold an association of a measurement and one of its functions. Next, a measurement called Average is declared, this could be any sort of measurement like the WindChill example from above. On the following line, the association of the

Average measurement and its function, Collect, are assigned to the variable MFP. Immediately thereafter, the pair is invoked from the variable MFP. Realistically, parameters would have to be passed, but they are ignored in this example. MFP is then assigned a new pair and then they are invoked. It is important to note that the invocation of the two pairs is handled identically from the point of view of MFP. No matter what association is in the variable MFP, the invocation is the same.

```
MeasurementFunctionPair MFP
Measurement Average
MFP = [Average, Collect]
MFP.Invoke
MFP = [Average, GenerateReport]
MFP.Invoke
```

Figure 2.3

Applying this system to the reporting problem involves creating a set of variable length lists of these associations. Each list is associated to a location in the code of an object. When, in the execution of an object's code, a list of measurement/function pairs is encountered, execution proceeds by traversing the list, invoking the function on the measurement while passing the object as a parameter. The pairs can be dynamically added or removed from lists.

This method limits the intrusive code to a reference to a list; in fact, the code is reduced to only one line. This is true regardless of the number and complexity of measurements that have been defined and placed on the list. Objects' source code needs to be modified only once to install the lists. While this requires the source code to be available, it is limited to one time. Once the lists have been installed, the source code never needs to be modified again, even when new measurements are developed. Newly developed reports need only be "hooked" to the appropriate lists. This is a very generalized method of dealing with measurements and reports.

2.2.5. List Deployment Considerations

In the first reporting method discussed above, global variables were used to allow access from anywhere. This implies that the intrusive code for a given measurement or report could reside anywhere. Hopefully, the most appropriate sites would have been selected for the intrusive code. Implementation of the measurement/function lists will require a limited number of sites to be used. Choosing these sites necessitates some heuristics.

There are key sites in the code of a program that many reports or measurements might find appropriate for data collection. Before and after loops of a major function are often good places for list of measurement/function pairs. Monitoring a process on each iteration of a loop is also useful. Generally, lists should be assigned to locations before and after critical sections of code. There is no guarantee that these generalized locations will be appropriate for all conceivable measurements, however, the majority can function from those places. If no appropriate site is available for a new measurement, it is not excessively difficult to assign a new list to the location. Careful selection of list locations will prevent this from happening often.

2.3. An Implementation

The implementation of this scheme requires the exploitation of the inheritance facility of an object oriented language. The entries in the list of measurement/function pairs must be treated uniformly. There is no way for the program to know at compile time what specific measurements might be in a list at any given point. If all measurements are derived from the same base class, they can be treated in a standard way.

The generic measurement base class should define the interface for all measurements derived from it. The process of measuring and reporting can be broken into four steps discussed above. These four steps make an ideal interface for a generic measurement. Measurements derived from the base class will make their own definitions of the four stages. Since measurements and their reporting functions are assigned to lists as a pair, the proper combination is assured.

The C++ implementation of this method uses three basic classes: Probe, ProbeAction and ProbeList. Probe is the base class from which all measurements and reports are derived. ProbeAction is a class that defines the association of a specific probe and one of its functions. It serves as a container to hold one association. Collects of ProbeActions are kept in instances of the ProbeList class.

Figure 2.4 shows the C++ definition of the class Probe.

```
class Probe
{
  protected:
  char ProbeName[30];
  FILE* OutputDestination;

  public:
  Probe (void);

  virtual void Reset (void*) {}
  virtual void Collect (void*) {}
  virtual void Calculate (void*) {}
  virtual void ReportHeader (void*) {}
  virtual void Report (void*) {}
  virtual double GetProbeValue (void*) { return 0.0; }

  inline char* GetName () { return ProbeName; }
};
```

Figure 2.4

The class Probe does not stand alone. It is a template from which useful measurements can be constructed. Only the derived classes will need to define the variables necessary for the calculation of a given statistical measurement. This generic class needs to define only the fields that every measurement will need. The class Probe defines two variables. These are the ProbeName and OutputDestination variables. These variables are useful in generating printed reports. The ProbeName is a string of characters that could be used to name the measurement. "Average Temperature" or "Wind Chill Factor" are examples. OutputDestination is a pointer to an output file to which reports may be sent.

A probe derived from the base class defines the virtual functions as is it deems appropriate for the measurement. These functions are directly related to the four stages of reporting. Reset is the function for the first stage of the reporting process. The function Collect is the representative for the second stage. Calculate processes the collected data for the third stage. Finally, ReportHeader and Report fill out the fourth step. The additional function GetProbeValue allows the probe itself to be monitored by another probe. The function allows the second probe to collect data from the first.

Each of the virtual functions is passed a pointer to the object that invoked the measurement/function pair. This is the mechanism that allows a probe to collect data. Probes can be constructed to be highly specialized. Typically, the Collect routine for a probe will assume that the pointer passed to it refers to an object of the type that it is expecting to monitor. It simply uses this pointer to invoke any of the public member functions of the object it is monitoring. In this fashion, it can collect its data about the object it is monitoring.

In the following example of figure 2.5, a complete definition of a probe is created. The probe's purpose is to monitor some other class that represents a thermometer. The thermometer class is responsible for determining only the current temperature. The probe, called AverageTemperature, gathers information from the thermometer in order to calculate an average temperature over some period.

```
class AverageTemperature : public Probe
{
  double RunningTotal;
  int NumberOfCollections;

  public:
  AverageTemperature (void);

  void Reset (void*);
  void Collect (void*);
  void Report (void*);
}

AverageTemperature::AverageTemperature (void)
{
  strcpy (ProbeName, "Average Temperature");
  OutputDestination = stdout;
}

void AverageTemperature::Reset (void*)
{
  RunningTotal = 0.0;
  NumberOfCollections = 0;
}

void AverageTemperature::Collect (void* t)
{
  RunningTotal += ((Thermometer*)t)->GetCurrentTemperature();
  NumberOfCollections++;
}
```

```
void AverageTemperature::Report (void*)
{
  fprintf (OutputDestination, "Average Temperature is : %lf",
                    RunningTotal / NumberOfCollections);
}
```

Figure 2.5

The functions that represent the stages of reporting are defined completely by this class. The Reset function sets the internal counter and sum variables to zero. The Collect routine queries the thermometer for the current temperature. The Report function prints the average temperature. Because the actual calculation of an average is trivial, it is incorporated into the printing of the report instead of being placed in a separate routine. The Calculate function defined in the base class is not used by this probe.

The association of a Probe and one of its functions is a measurement/function pair. They are represented in a list as an instance of the class ProbeAction. Inheriting the properties of a LinkNode enables an instance of ProbeAction to be a member of a linked list. Details of the linked list implementation are hidden and are unimportant to class ProbeAction. The C++ definition of the class ProbeAction is shown in figure 2.6.

```
typedef void (Probe::*ActionFunction)(void*);
class ProbeAction : public LinkNode
{
  Probe* P;
  ActionFunction Action;

  public:
  ProbeAction (Probe*, ActionFunction);
  inline void TakeAction (void* o) { (P->*Action)(o); }
  void Print (FILE*);
};
```

Figure 2.6

ProbeActions hold a pointer to the probe that is to be used, as well as a pointer to one of the probe's member functions. When traversing a list of

ProbeActions, the member function TakeAction is called for each ProbeAction found. This has the effect of applying one of four stage reporting functions to the probe. A pointer to the object that is being monitored is passed to the reporting function as a parameter. The probe then takes whatever actions dictated by its reporting function.

```
class ProbeList : public LinkedList
{
  public:

  void TakeProbeActions (void*);
  void Insert (Probe*, ActionFunction);
};
```

Figure 2.7

A ProbeList is a list of ProbeActions. It inherits almost all of its properties from the base class LinkedList. It adds only two new functions. TakeProbeActions traverses the list of ProbeActions invoking the TakeAction function for each in turn. Insert enters a pointer to a probe and a pointer to one of the probe's reporting functions as a ProbeAction onto the list.

Techniques for management of ProbeLists inside classes is flexible. Typically, a class that is to be monitored will have several ProbeLists associated with it. Each list will represent a certain location in the code for the monitored class. The target class defines how many ProbeLists there are and where they are used. It is the responsibility of the class being monitored to invoke the TakeProbeActions function for a given list at the appropriate time. This is the intrusive code outlined earlier.

The act of assigning a measurement/function pair to a specific list is also the responsibility of the class being monitored. A member function must be created to take a measurement/function pair and assign it to the appropriate list. An array of lists can be maintained with each element being a list associated with a specific location in the code. Assigning a measurement/function pair to a specific location is as simple as providing the index to the array that corresponds to the list for the desired code location. Constants setup with appropriate names can make the code more readable. An example follows in figure 2.8.

```
const int BEFORE_TEMPERATURE_RETRIEVAL = 0
const int AFTER_TEMPERATURE_RETRIEVAL = 1
...
AverageTemperature AT;
Thermometer T;
T.AssignProbe (AFTER_TEMPERATURE_RETRIEVAL, &AT, &Probe::Collect);
```

Figure 2.8

In this example, two objects are defined: a Thermometer object called T and a probe to calculate the average temperature called AT. Assume that the thermometer class acquires its temperature through some method for which there is a "before" and "after" phase. Two ProbeLists exist in the Thermometer class. The first list is traversed in the "before" phase and the second is traversed in the "after" phase.

The last line of the example invokes the member function of T called AssignProbe. This takes a pointer to the probe (&AT) and a pointer to the reporting function (&Probe::Collect) and places them into a list as a measurement/function pair. The ProbeList itself handles the conversion of the measurement/function pair into a ProbeAction.

```
ThermometerProbeLists[BEFORE_TEMPERATURE_RETRIEVAL].TakeActions (this);
... do whatever necessary to acquire temperature ...
ThermometerProbeLists[AFTER_TEMPERATURE_RETRIEVAL].TakeActions (this);
```

Figure 2.9

These two lines represent the intrusive code inside the Thermometer class. The probe designed to report on the average temperature will collect its data whenever the last line of example above is executed. "This" in the parameter list of the TakeActions call is a pointer to the instance of the Thermometer class (T).

The other reporting functions of the AverageTemperature probe do not have to be assigned to this class. The Thermometer class may be part of a larger scheme controlled by an overall WeatherStation class. The Reset and Report functions of the AverageTemperature probe might, more appropriately, be placed in ProbeLists within that class. There are no restrictions preventing a probe's reporting functions from being assigned to several classes. A single function may be assigned to several classes. It is vitally important to endow the function with the ability of differentiating the classes that might be passed to it.

2.4. Other Uses of Probes

Probes are very flexible constructions. Their ability to take part in the on going process of a program makes them useful for more than just reports.

2.4.1. Debugging

Programs such as the genetic algorithm or simulations may take many hours to run. There may be little or no output during the process. The debugging stages of development for programs such as these can be extraordinarily difficult.

Traditional debuggers are not very helpful because they require constant human interaction. They may be able to report on the values of variables at certain points in the execution, but few debuggers can detect relationships between variables. For example, a debugger can detect out of range values of a variable and stop a program at the point of the exception. However, if the definition of "out of range" is dependent on other variables is the program, a traditional debugger is useless.

It becomes necessary to insert code into the program to detect subtile relationships between variables. When an inappropriate relationship is detected, the inserted code must stop the program and report the problem. After the insertion of several sets of error detecting code, the original code becomes cluttered.

This is exactly the problem that Probes can alleviate. The creation of a probe to test for the subtile relationships is straightforward. Any number of probes can be added with minimal modifications to the original code. When the probes are no longer needed, they can be easily removed. If they should become needed again in the future, adding them back in is simple.

2.4.2. Active Probes

Because probes can call any of the public methods of the object they are monitoring, the door is open for probes to take a more active role. It is possible for a probe to call a routine that will cause the object it is monitoring to take an action.

Consider a probe that is monitoring the WindGauge object of a weather station. It can collect the current wind speed information. If the wind speed exceeds a certain threshold, the probe can order the wind gauge to shut itself down, avoiding damage.

2.5. Conclusion

Probes implemented as defined above are clearly very flexible. A probe can be defined to take either active or passive roles. The framework to support probes in existing code requires minimal modification. Simply adding ProbeLists at key locations in the code as well as an assignment function is sufficient. Once the framework is in place, new probes can be developed, tested and put into use without any recompilation of the code that the probe monitors. Probes minimize the amount of work necessary in the development of new measurements and reports.

3. Experiment Control - Iterators

In many types of programs, especially simulations, it is desirable to allow the user to vary the program's controlling parameters at will. This facilitates experimentation by allowing the user to explore different combinations of parameters. Differing combinations may induce differing behaviors from the program which in turn can produce an insight about the real world system that the program models. The process of exploring the behavior of a model under varying conditions is a vital tool for developing an understanding of the real world system.

The flexibility of a simulation in allowing an experimenter to explore, is often synonymous with the power of the simulation. The techniques used to construct the simulation define the flexibility of the model. If the simulation program is built in a rigid manner that does not allow the user to vary the controlling parameters, the program clearly is not very flexible.

Consider a model of the population dynamics of an ecosystem. The controlling parameters could consist of the initial population sizes of the predator and prey species along with their reproductive rates. The user could explore the model by repeatedly invoking the program and providing varying values for the four controlling parameters. With each execution of the program the user would dutifully enter the new information and record the results.

If the execution time of the simulation is long, then the cycle of entering new information and recording results becomes very tedious. The user might have to sit and wait between iterations. If the user could specify a series of values for the program's controlling parameters and the program could execute them one at a time without user intervention, the program would be more flexible.

Simulation programs are often dynamic programs. As the process of exploring the model proceeds, shortcomings of the program can become apparent. New controlling parameters may have to be added to make the program more accurate or useful. Another measure of a simulation's power is how easily these parameters can be added. Clearly, the simple ecosystem model mentioned above will exhibit shortcomings very quickly if it only takes into account one type of predator and one type of prey.

The problem is to develop a system for controlling a program that facilitates both the user in the process of experimentation and the programmer in the process of enhancing the program. The focus here is on programming structures that allow for the easy addition of flexible user controls. How the parameters are used within the simulation are outside the bounds of this discussion.

3.1. Nested Loops

Loops are the first method that comes to mind when the problem is to vary the value of a parameter over a range. Nested loops are a familiar and standard method for accomplishing this. Each variable that is to be controlled is assigned a nesting level. The loop code is then constructed to reflect the nesting level and the starting value, ending value and increment for each variable.

In the ecosystem example, the parameters that need varying might be the initial population sizes for lions, gazelle, zebra and giraffes. The user would be prompted for a range of these initial values along with an increment. A loop would control each parameter.

```
prompt for LionsLow, LionsHigh and LionsInc
prompt for GazelleLow, GazelleHigh, GazelleInc
prompt for ZebraLow, ZebraHigh and ZebraInc
prompt for GiraffeHigh, GiraffeLow, GiraffeInc
for (Lions = LionsLow to LionsHigh; inc Lions by LionsInc)
  for (Gazelle = GazelleLow to GazelleHigh; inc Gazelle by GazelleInc)
    for (Zebra = ZebraLow to ZebraHigh; inc Zebra by ZebraInc)
      for (Giraffe = GiraffeLow to GiraffeHigh; inc Giraffe by GiraffeInc)
```

```
LionPopulation.Size = Lions
GazellePopulation.Size = Gazelle
ZebraPopulation.Size = Zebra
GiraffePopulation.Size = Giraffe
EcosystemModel (LionPopulation, GazellePopulation,
                ZebraPopulation, GiraffePopulation);
```

Figure 3.1

This system works just fine. The user could specify the lion population to range from two thousand to three thousand by increments of five hundred. Similar ranges could be specified for the other animals. The program would then run the simulation for all combinations of the values specified for each animal. The user has to run the simulation only once to get many sets of output.

The first problem with this technique occurs when the user wants to run the program with a nonlinear set of values for a parameter. For example, the desired population size of lions may be 1000, 1500, 3000, and 5000. A normal loop can only handle this indirectly. The input routine must allow the user to input a list of values which are then stored in an array. The loop then iterates over the array. The values for population are picked out of the array based on the loop counter as an index.

The second problem comes from the user being unable to specify the order in which the experiments take place. The first invocation of the ecosystem model uses the first value from all the loops. The second invocation uses the first value of all but the innermost loop. The innermost loop is dedicated to the giraffe population and it is destined to cycle through all of its possible values before the outer loops advance to their next values. The user cannot change the nesting level in order to set, for example, the lion population to the innermost loop.

From the programmer's point of view, the classic nesting of loops is not a very flexible structure. The introduction of a new parameter to the program translates into the addition of a new level of nesting. It is undesirable to allow the nesting level to get too deep. This is not necessarily a subjective aesthetic limitation, some programming languages have limits on nesting depth.

The structure of nested loops is also too rigid in other ways. Consider the complications if the user is allowed to select which animals the simulation is to use out of a list of one hundred animal species. Each animal would have a parameter for the initial population size. A given animal could have additional parameters that no other animal might have.

It is clearly impractical to program a nesting of loops for every possible combination that could be selected by the user. It is also impractical to attempt to write nested loops to a depth that would be required if the user opted to use all the animals. If each animal were to have two parameters associated with it and there were one hundred animal species, there would be at least two hundred nested loops.

3.2. Loops as Data

A loop represents a series of values for a variable. The variable begins with the first value. At the end of the first iteration of the loop, the variable skips to the next value. This continues until the last value is used, whereas the loop terminates. The process is typically implemented using "for" loops. The series of values is calculated on each iteration of the loop. This ties the series of values tightly to the code of the for loop. There are other methods that liberate the values from the close bond with the code.

Dissecting the functionality of a "for" loop reveals that it is merely a variation of a "while" loop. It defines a pair of standard operations to sequence through a list. These operations are initializing the current value to the beginning of the series and advancing the current value pointer to the next value.

A loop can be represented as an association of two things: a list of values and a pointer to the current value. In addition, there can be a variable, called the target, that is to receive the series of values. Operations can be defined to act on an association that mimic the internal operation of a "for" loop. Then by using a "while" statement, the functionality of a "for" loop can be achieved without using one. Additional operations, beyond those of a "for" loop, can also be defined. This allows the association, the list of values and the current value pointer, to be divorced from the loop. The association becomes a parameter for a group of operations that implement a loop. Several operations can be defined that use an association: initialize, reset, next, test and get value.

The initialize routine fetches the values in the list. It could be done by querying the user or by some other method. The details are not important at this stage. The list itself can be implemented in any ordered manner: an array, linked list, et cetera.

The reset and next operations both concern the pointer to the current value in the list. The reset function would set the pointer to the first value in the list. The next operation moves the pointer to the next value in the list. If the end of the list is reached, the next operation should not wrap back to the first value.

The test function checks the position of the current value pointer. If the pointer has gone beyond the end of the list, this function should signal that the loop is completed.

The get value operation is most often used for assignment. It allows the target variable to use the pointer to get a value for itself. This operation is separate from the 'next' operation for flexibility reasons. The association between the target variable and the other two components of the loop is a one way association. The target variable "knows" about the loop from which it gets its values. The loop, however, does not "know" anything about the target variable. This allows for a greater degree of flexibility at a later stage in the implementation of this scheme. The target variable does not need to be consistently the same variable. Several variables can share the role of being the target. It is even possible that the target variable is allocated and deallocated within the body of the loop. With every iteration, there may be a new target variable.

```
LoopAssociation.Initialize = {2.89, 3.14, 6.023, -97.8}
LoopAssociation.Reset ()
do
  print LoopAssociation.GetValue ()
  LoopAssociation.Next ()
while LoopAssociation.Test () does not signal end-of-loop
```

Figure 3.2

This example demonstrates how this type of loop could work in a quasi-object oriented language. The association of the list and the pointer are represented by an object called LoopAssociation. The list is assigned a series of values in the first line. The pointer is then reset to the first value from the list. The body of the loop must be placed in a traditional looping structure, in this case, a do...while. The body of the loop consists of simply printing the current value of the loop. Here, target variable is unnamed. It can be considered to be some temporary location within the print function. The next function controls the while loop. It moves the pointer to the next value and the while loop repeats. If there is no next value to move to, the next function signals the end of the loop and the while loop terminates. The output of this example should be the list of four values.

Nested loops can be thought of as a set of series of values for a set of variables. When the body of the loop is completed, only the inner most variable changes to the next value. If the innermost variable was on its last value, then it resets back to its first value. The next outermost variable then proceeds to its next value. When the absolute outermost variable has used its last value, the nested loops terminate.

A nesting of loops can be constructed by linking several loop associations together in an ordered set. The set then can respond to a simple group of operations that will affect all of the loops internal to the set. The operations include: assign a single loop to the set, reset and step.

The assign function associates a single loop with the set. This operation must be invoked once for each loop that is to be part of the loop. By definition, the order in which the loops are added to the set directly corresponds to the nesting order of the loops. The first loop added to the set will be the outermost loop. The last loop added to the set will be the innermost loop.

The reset operation causes the set to traverse its list of loops telling each to reset itself. The result is to have all of the pointers in the various loops set their first values from their respective lists.

The step operation causes the innermost loop to proceed to its next value. It does this by invoking the next function for that loop. If the loop signals its completion, the reset function is invoked and the next outer loop's next function is invoked. This action progressively bubbles out to the outermost loop. When the outermost loop signals that it has completed, the step function signals the completion and the entire set of nested loops terminates.

The example in figure 3.3 nests three loops. The loops are set up in the first three lines. Assigning the loops to the set in the following three lines places Loop1 in the position of outermost. Loop3 serves as the innermost loop. All three loops are reset by resetting the set in the next line.

```
Loop1.Initialize = {2.89, 3,14, 6.023, -97.8}
Loop2.Initialize = {14.0, 15.0}
Loop3.Initialize = {1, 2, 3, 4}
LoopSet.Assign (Loop1)
LoopSet.Assign (Loop2)
LoopSet.Assign (Loop3)
LoopSet.Reset
do
  print Loop1.GetValue(), Loop2.GetValue(), Loop3.GetValue()
while LoopSet.Step() does not signal end-of-nested-loops
```

Figure 3.3

A single do...while loop controls the entire set of loops. Since all the loops have been reset, the first iteration causes the values 2.89, 14.0 and 1 to be printed. The invocation of the loop set's step function, in the condition of the while loop, causes the innermost loop to proceed to its next value. The print line then produces the output 2.89, 14.,0 and 2. The process will proceed until the final output of -97.8, 15.0 and 4. At this point, the step function signals the end of all the loops and the do...while loop terminates.

In the example above, any number of loop associations could have been given to the set. The number of loop associations has no bearing on the format of the do...while loop.

This technique addresses all of the problems encountered in the ecosystem example. Because the series of values that a loop association uses is stored in a list, a non linear set of values can be specified. The last example demonstrates this.

Since this method treats the loop associations as data, the associations can be manipulated like any other piece of information. This means that they can be created and assigned to a set dynamically. In the ecosystem example, the user was allowed to select any animal from a list of many species. As the user selects each species, the appropriate number of loop associations could be created and assigned to a set. The user, at this point, could be prompted for the values for the series in the list. The set will have exactly the correct number of loop associations. The nested loops have been set up without any need for reprogramming.

Devising methods of rearranging the order of the nesting are not difficult. The user can be given the opportunity to sort the set in any manner. A function could be added to the set that would allow the user to prioritize the loop associations. In a graphically based program, this could be done visually by rearranging icons that represent the nesting levels. Alternatively, the user could couple a number with each loop association. The number could be interpreted as the nesting level number. Sorting the set by the number before use would effectively change the nesting.

3.3. The User Interface

The objective of this method of handling loops is to make the controlling parameters of a simulation friendly and powerful for both the programmer and the user. The workings of loops, as they have been suggested here, provide a framework for a flexible system. Since the loops are treated in a uniform manner, it is only natural that the user interface of these loops are also consistent.

The initialization step of a loop association calls for the list to be given a series of values. This means that the user is to provide the values. It is important to provide suitable prompting to illicit a correct response from the user. The initialization step should provide facilities to prompt the user with an appropriate phrase, as well as inform the user if there are any default responses.

If the user responds to the prompt with just a carriage return, the series should take on the default value. Alternatively, the user can type a series of values. To make a long linear series easier, the user should have the option of specifying a low value, a high value and an increment.

```
Enter the initial population of lions [1000]: <cr>
Enter the initial population of gazelle [10000]: 8000 8500 8750
Enter the initial population of zebra [5000]: (4000 8000 1000)
Enter the initial population of giraffe [3000]: (2000 4000 500) 5000 6000
```

Figure 3.4

The computer prompts the user with the underscored text in this example of Figure 3.4. On the first line, the simulation wants the number of lions in the initial population. It specifies that the default is 1000. The user responds with just the carriage return which accepts the default. In the second line the default for the initial population of gazelle is 10000. Instead of accepting this value, the user specifies three values. The corresponding loop association will initialize its list to the values specified by the user. On the third line the user specifies five values instead of accepting the default single value of 5000. In the parenthesized form specified by the user, the first value is the low; the second value is the high; the third value is the increment. In the loop associations list, the input translates to the five values: 4000, 5000, 6000, 7000, 8000. On the last line, the user combines discrete values with an iterated group. The values used by the loop association will be: 2000, 2500, 3000, 3500, 4000, 5000, 6000. The user may specify any combination of iterated groups or discrete values.

3.4. Implementation

Object oriented programming provides many of the tools necessary for the simple implementation of this scheme. Loops need not be restricted to floating point values. Through the OOP features of inheritance and virtual functions, loops can be expanded in type to include integers, long integers and strings.

The loop association base class defines the interface for all types of loops. Regardless of the actual data type used by the loop, every loop association needs to be able to respond to the five basic loop functions: initialize, reset, next, test and get value. The base class also defines the components of a loop that are common to all types. This includes the variables necessary for implementing the user interface.

3.4.1. The Base Class Iterator

```
class Iterator
{
  char DefaultString[MAX_INPUT_LENGTH];
  char InputString[MAX_INPUT_LENGTH];
  char Prompt[MAX_INPUT_LENGTH];
  int NumberOfValues;
  int CurrentPosition;

  public:
  Iterator (void);
  void GetIterator (char *, char *, FILE*);
  inline int IsDone (void);
  inline void Next (void);
  inline void Reset (void);
```

```
        ...
        virtual void ParseInput (char*) = 0;
        ...
      }
```

Figure 3.5

Because the iterators derived from this base class implement loops using different variable types, it is the derived types' responsibility to define the list containing the series. However, the current value pointer can be implemented in the base class because it is simply an index into the list. The variable CurrentPosition serves in that capacity. The variable NumberOfValues, keeps track of the length of the series.

The three character strings defined at the beginning are used to implement the user interface. When an iterator is created, a prompt and default are specified. These will be presented to the user during the iterator's initialization process. The exact input given by the user is retained in the character array UserInput. The input is kept so that it can be used as output in a report. This could allow experiments to be repeated without retyping the input, in addition to verifying the input during an audit procedure.

The routine GetIterator serves as the initialization of the iterator. It sets the values for the default and the prompt. It then uses them to get input from the user. Because each derived iterator uses a different data type, it is the derived iterator's responsibility to parse the input from the user. This is done with the ParseInput routine. All derived types of iterators must define their own ParseInput routine. Ideally, each ParseInput routine should allow the same form of input. In other words, all types of iterators should allow the user to input discrete values as well as iterated groups. However, there is no facility to enforce this.

The Reset function simply returns the CurrentPosition variable to the beginning of the list. The Next function increments the CurrentPosition. IsDone compares the CurrentPosition with the NumberOfValues. If CurrentPosition is beyond the end of the list, IsDone returns a true condition.

The GetValue function is not defined by the base class. This is because each derived class will return a value of a different type. Therefore each derived class is responsible for defining its own GetValue function.

IntIterator in figure 3.6, defines a typical derived iterator. Its purpose is to allow the user to iterate over a series of integers. This is the iterator that would be chosen for the initial population size of an animal in the ecosystem example.

```
class IntIterator : public Iterator
{
  int Values[MAX_NUMBER_OF_VALUES];

  public:
  IntIterator (void);
  int GetValue (void) { return Values[GetCurrentPosition()]; }
  virtual void ParseInput (char *);
  ...
};
```

Figure 3.6

The IntIterator provides an array of integers to serve as the list of values. The GetValue function returns the value of a position in the array using the CurrentPosition index from the base class. The iterator also defines the mandatory ParseInputFunction.

3.4.2. IteratorSet

The IteratorSet manages a list of Iterators. By maintaining a list of Iterators, the class is able to implement all the features of dynamically nested loops.

```
class IteratorSet
{
  FILE* Source;
  Iterator* Iterators[50];
  int NumberOfIterators;

  public:
  IteratorSet (FILE*);
  Iterator* AllocateIterator (int, char*, char*);
  void Reset (void);
  int Step (void);
  void PrintIterators (FILE*);
  void Save (void);
  ...
};
```

Figure 3.7

The list of Iterators is stored in the form of an array of pointers called Iterators. The NumberOfIterators keeps track of the nesting depth.

To add an iterator to the set, the AllocateIterator function is used. It must be provided with a flag identifying the type of iterator desired, the prompt and the default value. The process of creating iterators invokes the iterators' user interface. If the variable Source is pointing to the standard input (stdin), the user is prompted for the series for the new iterator. If the variable Source instead points to a file, the series of values is read in from the file. In either case, the function returns a pointer to the newly created iterator. The object that requested the allocation of a new iterator will then use the pointer to retrieve the series of values.

The Save function allows the user the option of saving the responses given to the iterators' prompts. An output file must be specified if the user chooses to save. The IteratorSet then cycles through its list of iterators causing each to output the response that was given to it.. The file saved in this manner is in a form that can be used as input in the iterator allocation process.

Once all of the iterators have been created, the nesting of loops is ready to use. The Reset function will cycle through the list of all the iterators causing each to reset to their first value. The Step function causes the iterators to proceed to the next values in the manner outlined above. If the outermost iterator has used its last value, the Step function returns a false condition indicating that the loops are complete.

```
void Lion::SetupIterators (IteratorSet* IS)
{
  InitialPopulationIterator = (IntIterator*) IS.AllocateIterator
                              (INT_ITERATOR, "Initial Population Size",
                               "1500");
  AttritionRateIterator = (DoubleIterator*) IS.AllocateIterator
                          (DOUBLE_ITERATOR, "Attrition Rate by Disease",
                           ".40"
  ReproductiveRateIterator = (DoubleIterator*) IS.AllocateIterator
                             (DOUBLE_ITERATOR, "Reproductive Rate", "1.2");
}
```

Figure 3.8

The ecosystem simulation provides a good example of the IteratorSet's use. The population of lions must iterate over three values: the initial population size, attrition rate due to disease and reproductive rate. When the user selects lions to be part of the ecosystem, the class lion requests three iterators from the iterator set. It provides prompts and defaults for each of the iterators and then saves the pointers it receives.

```
void Lion::GetNewValues (void)
{
  PopulationSize = InitialPopulationItertor->GetValue();
  AttritionRate = AttritionRateIterator->GetValue();
  ReproductiveRate = ReproductiveRateIterator->GetValue();
}
```

Figure 3.9

At the beginning of each iteration of the iterator set, the objects that have allocated iterators must be allowed to retrieve new values. In this example, the class Lion has a function specifically for getting the new value of its three iterated variables. For this example consider this function to be repeated for each animal class that was available for the user.

```
IteratorSet IS(stdin);
...
ListOfAllSelectedAnimals = AllowUserToSelectAnimals();
for each animal i in ListOfAllSelectedAnimals do:
  i->SetupIterators();
...
IS.Reset();
do
{
  for each animal i in ListOfAllSelectedAnimals do:
    i->GetNewValues();
  EcosystemModel (ListOfAllSelectedAnimals);
} while (IS.Step());
...
```

Figure 3.10

This pseudo-C + + code fragment demonstrates a complete dynamic nesting of loops. The IteratorSet is created on the first line, specifying that input is to come directly from the user instead of a file. A list of animals to be included in the simulation is the created by the user. The SetupIterators function is called for each of the animals selected. This causes the user to be prompted for whatever series values are needed for each animal. This allocates new iterators, which are placed in the iterator set. The iterator set is then reset and the nested loops begin. The first step in the loop is retrieving values from the iterators. The ecosystem simulation is then given the list of animals with their current values. When the simulation ends, the IteratorSet steps to the next value and the loop repeats.

3.5. Conclusion

Iterators and IteratorSets define a very flexible method of implementing loops in a simulation program. Not only do they allow loops to be created dynamically, but they also allow for dynamic nesting. This solves the inflexibility problems found with tradition looping structures. The user interface aspects of Iterators create a complete unified package that can be used to control a simulation or a similar program.

4. The Genetic Algorithm

OBJGEN is an object oriented program for use in experimentation with genetic algorithms for optimization. It is based on a framework designed to provide maximum flexibility as an experimental platform. The program and its framework are malleable; many different variations of genetic algorithms can be constructed with a minimum amount of recoding. Iterators, Probes and the hierarchical inheritance structures of object orient programming provide much of the flexibility.

4.1. Genetic Algorithm Background

Binary Genetic Algorithms are search algorithms that are based on the mechanics of natural selection and natural genetics [1]. Given a parametric equation to be optimized (the objective function) and a set of bounds on the parameters, a genetic algorithm will search for a global optimum. Genetic algorithms have been shown to be effective over a wide variety of different optimization problem types. In addition to real parameter optimizations, genetic algorithms have also been applied to combinatorial problems.

Genetic algorithms begin with a set of randomly chosen points in the search space. Each point is considered an individual member of a population. Through the processes of the algorithm, a population will spawn a succeeding generation. Subsequent populations will begin to converge to one area of the search space.

A typical example of a genetic algorithm is the one embodied in the GENESIS program [2]. GENESIS can be used for optimization of parametric functions with real parameters. Internally, the position of a point in the search space is encoded as a binary string. There is one section of the binary string for each parameter of the objective function. The binary string can be thought of as the analog of DNA in biological genetics. The binary string is the genetic material that holds all the information that defines the individual's position in the search space.

Each individual in the initial population is rated on performance. This is done by decoding the bit string and then applying the point it represents to the objective function. The objective function returns a floating point value which is then associated with the individual as the individual's performance.

Comparison of the performances of individuals in a population determines the fittest individuals. These best individuals are then selected to form a new population. Individuals that were not selected do not survive. Metaphorically, this is the survival of the fittest as defined by Darwin. The algorithm used by GENESIS to effect the selection is a nondeterministic technique called the Baker selection method [3]. A selected individual will generate a number of offspring in direct proportion to its relative fitness compared with the other selected individuals. In other words, if the normalized fitness of an individual is twice that of another individual, the first individual with have on average twice the number of offspring in the succeeding generation.

Selection pressure is only one aspect of the process of advancing from one generation to the next in biological evolution. Genetic algorithms usually simulate mutation and mating in addition to selection.

Mutation acts on the binary string. A threshold known as the mutation rate controls mutation. The threshold represents the probability of a given bit in a given string will be inverted. The inversion of a bit has the effect of moving the individual to a new point in the search space.

The exchange of genetic material between two individuals in a genetic algorithm is known as crossover. In the metaphor with biological evolution, this is the mating of two individuals to produce offspring. Unlike the biological model, however, the union of two genetic algorithm individuals always produces two offspring and destroys the parents. Two positions randomly chosen in the bit string determine a segment of genetic material that is exchanged between the two parents. When the exchange occurs, the individuals that were the parents become the offspring. In effect, crossover forces the population to try different combinations of parameters.

Crossover knows nothing of the boundaries between the parameters in the binary string, thus an exchange of genetic material is not simply an exchange of parameters between individuals. When a crossover point falls inside a parameter, the effect is a perturbation of the parameter in both individuals [4].

A trial is an invocation of the objective function on an individual that has not been tested in the past. Any individual that is changed by mutation or crossover is considered a new untried individual and is therefore applied to the object function for evaluation and counted as a trial. If an individual survives into a succeeding generation and manages to get by mutation and crossover with no changes to its binary string, it is not applied to the objective function and therefore a trial has not occurred. The number of trials can be thought of as roughly the number of points in the search space that have been examined.

Once selection, mutation and crossover have acted on a population the process of advancing to the next generation is complete. The algorithm repeats until a stopping criteria had been met.

Without knowing the global optimum beforehand, it is very difficult to determine if a population has converged. It is necessary to devise other methods to stop the algorithm. One method uses the concept of trial. At the start of an experiment a maximum number of trials is selected. When the maximum is achieved, the algorithm stops. This same idea can be applied to the number of generations. If several generations go by with no changes in the number of trials, the algorithm stops. This is called the spinning threshold.

Several measurements of the performance of genetic algorithms have been devised. Off-line and On-line performance measures rely on the concept of a trial. Off-line performance is geared toward measuring convergence while on-line measures ongoing performance [1].

4.1.1. Real Vector Genetic Algorithms

In an RVGA, the encoding of the parameters into binary strings is bypassed [4, p8]. Each individual now holds a vector of real numbers that are used as the parameters for the objective function. In theory this variation on a genetic algorithm may be more efficient because it eliminates the overhead of the binary string conversions. The differences in the representation of the "genetic material" necessitate the redefinition of several of the processes of the genetic algorithm.

Mutation is redefined to act directly on the points of the population rather than the encoding of the points. The mutation rate can now be considered the probability that a parameter will be perturbed by mutation. The actual process of mutation can be done in several ways. One possibility is to simply allow a new value to be selected for a given parameter in a range centered on the old value of the parameter. The size of the interval can be determined at run time with a setting called MutationSize expressed as a percentage of the entire range of possible values for a parameter. Another alternative that would facilitate comparison of BGA's with RVGA's would be to simulate mutation of binary individuals.

Crossover is redefined to be an exchange of parameters between individuals. For example, if individual one has parameter values ABCD and individual two has values EFGH a crossover might produce two new individuals with parameters of AFGD and EBCH. Unlike crossover in BGA's, crossover in RVGA's cannot take place in the middle of a parameter. This means that crossover in a RVGA does not perturb parameters.

All other techniques involved in genetic algorithms remain the same for RVGA's. The selection method as well as the measures of off-line and on-line performance do not need to be changed.

4.2. The Framework

The OBJGEN program consists of an implementation of the genetic algorithm using the object oriented language C++. The program is based on a framework that is independent of the type of genetic algorithm implemented in the program. The framework provides the flexibility and much of the power of the program.

There are five major families of classes that make up OBJGEN: Individuals, Populations, Genetic, Iterators, and Probes. In addition there are three auxiliary classes that serve in supporting roles: FileManager, FunctionDispatcher, and IndividualManager.

The framework provides a skeleton of a genetic algorithm. All of the major functions of the algorithm are implemented in a generalized manner. This allows the framework to function with a wide number of variations of genetic algorithms. Part of the flexibility of the program stems from the fact that new features can be added without rewriting or recompiling the framework.

4.2.1. Individuals

Individual is the base class from which all types of individuals are derived. This means that Individual defines the basic attributes of individuals without defining the implementation details. The classes derived from Individual are responsible for filling in the details. For example, an individual that is used in a genetic algorithm for solving a combinatorial problem will have its genetic material represented in a manner appropriate for combinatorial problems. An individual from a genetic algorithm that optimizes parametric objective functions with floating point parameters would have a different representation. Both types of individuals would derive from the same ancestor using the object oriented feature of inheritance.

There are two constructors for this class, each is used in a different situation. The first constructor is used to create the prototype individual. In the creation of this individual, the user is prompted, through the use of iterators, for key values that control features of the individual.

The prototype individual's sole purpose is to spawn new individuals. The prototype individual is given to the class that represents a population. The population fills itself by cloning the prototype individual. In this manner, a population is independent of the type of individuals it carries. Any derivation of an individual can be given to a population and the population can successfully use it.

The process of cloning an individual uses the second constructor. The second constructor queries the iterators rather than the user for the controlling values for the individual's features.

The class Individual, shown in figure 4.1, declares the interface that is common to all types of individuals. The interface consists of functions though which all actions involving individuals are routed. It does not define the details of the implementation of the interface, it simply declares that these routines will exists for all types of individuals. Each type of individual is required to define in detail the implementation of these routines. A given type of individual may add new routines to the interface. The base class Individual merely states that the routines it defines are the bare minimum that a derived type of individual must have.

```
class Individual
{
  static FunctionDispatcher* FD;
  static IteratorSet* IS;
  static int FunctionNumber;
  static Iterator* IndividualIT;

  static int Usage;

  int NeedsEvaluation;
  double Performance;
  ...
  public:
  ...
  virtual int  CalcPerformance (void) = 0;
  virtual Individual* Clone (void) = 0;
  virtual void Copy (Individual*);
  virtual void CrossOver (Individual*) = 0;
  virtual void Mutate (void) = 0;
  virtual void Print (FILE*);
  virtual int  operator== (Individual*) = 0;
  virtual void RandomSetup (void) = 0;
}
```

Figure 4.1

The CalcPeformance routine is intended to cause an individual to invoke the evaluation function. The evaluation function is passed the individual's genetic material and then return a single floating point number representing the individuals performance. The individual must then store that value.

The Clone routine creates a new individual, based on the type of the current individual, using the second constructor. The genetic material of the current individual is not copied. The routine results in a brand new "empty" individual.

The Copy routine is very simple. It copies the genetic material of the individual passed to it. This enables an individual to become an identical twin of another individual.

The CrossOver routine implements the crossover operation between two individuals. The individual passed to this routine and the current individual will exchange genetic material in a manner appropriate for the individuals' type.

Mutate is the routine that accomplishes the mutation operation.

Print is used to dump the contents of an individual to an output device or file. The print destination must be passed to this routine in the form of a pointer to a file.

Operator= = is a routine that compares the genetic material of two individuals. If they contain the same information this routine returns a one, if they are different it returns a zero.

RandomSetup sets an individual's genetic material to a random state. This routine is used at the beginning of an experiment to initialize an individual.

In addition to defining the interface for individuals, the class Individual holds information that is common to all individuals. Regardless of the variant of individual in use, there are properties that all individuals have.

Individuals must be able to be evaluated; therefore each individual must hold a value representing its own performance. Each individual has its own variable to hold its performance. Appropriately, this variable is called Performance. The flag, NeedsEvaluation, is used to signal if an individual's performance needs updating. If crossover or mutation results in a change in an individual's genetic material, the individual needs reevaluation. It is the responsibility of the individual's CrossOver and Mutation routines to set this flag.

Each instance of an individual has its own copy of the variables NeedsEvaluation and Performance. There are five variables for which there is only one copy. These variables are common to all instances of Individual.

FD is a variable that holds a pointer to a function dispatcher. A function dispatcher is an object that manages a list of objective functions. When an individual needs access to an objective function to evaluate itself, it simply asks the function dispatcher for the function. The function dispatcher then returns the appropriate objective function in the form of a pointer. The individual then can invoke the objective function via the pointer.

An individual informs the function dispatcher which objective function it needs, through the use of the FunctionNumber variable. This variable is controlled by the user via an iterator.

The variable IndividualIT is a pointer to the Iterator that controls the FunctionNumber variable. When a new group of individuals is created, the first individual makes an inquiry to this Iterator. The Iterator returns the number of the objective function that will be in use during the current experiment. The individual in turn will use this value to get the objective function from the function dispatcher.

The pointer IS is a pointer to an IteratorSet. The IteratorSet is an object that controls all of the Iterators in the program. Its value is set during the creation of a prototype individual. When a new iterator is needed, it can be allocated by a request to the IteratorSet. This value is not used, except in the creation of the FunctionNumber Iterator in the first constructor. Its presence is merely a convenience to facilitate future expansion.

Usage is a counter that keeps track of the number of individuals that have been created. The prototype individual is excluded from this count.

4.2.1.1. RealParameterIndividuals

The class RealParameterIndividual, shown in Figure 4.2, is an intermediate level in the implementation of the individuals in the RVGA. It is a class derived from the base class Individual and it therefore inherits all of the properties of that class. As a derived class, it has the responsibility of defining the details that are left out of the base class. This includes the actual representation of the genetic material and all of the interface routines that the base class declared but did not define. Because this class is an intermediate level, it can pass the responsibility for definitions of the interface routines on to classes that will be derived from it.

```
class RealParameterIndividual : public Individual
{
  static DoubleFunctionPointer Evaluator;
  static double *RealMinimums;
  static double *RealRanges;
  static int NumberOfParameters;
```

```
        static double MutationRate;
        static Iterator *RealIndividualIT[2];
        static int Usage;

        double *RealParameters;
        ...
    }
```

Figure 4.2

The genetic material of an instance of class RealParameterIndividual consists of a vector of floating point values. The size of the vector is dependent on the number of parameters needed by the objective function. This vector is called RealParameters and is implemented as a pointer to a double. Memory for the vector is allocated at the time of the creation of the individual in the second constructor. RealParameter is the only instance variable in this class.

All other variables are class variables. That means there is only one copy of these variables used by all instances of the class. The values of these variables need to be set only once at the beginning of an experiment. The Usage variable controls this. This variable counts the number of RealIndividuals that have been created. Only during the first one's creation will constructor execute the code to initialize the values of the class variables.

Evaluator is a class variable that holds a pointer to an objective function. This is the objective function that is returned by the function dispatcher. This variable exists for speed considerations. It is certainly possible to go through the function dispatcher whenever the objective function is required. However, the overhead of doing so would cause the program to slow considerably. A simple solution is get and store the pointer to the function once.

The Evaluator variable is stored in the RealIndividual class instead of the base class for flexibility reasons. Pointers to functions in C++ are typed according to their return value and their parameters. These together are called the signature of a function. The RealIndividual class is designed to work with functions that have floating point parameters. Placing this value in the base class would restrict all derived classes to working with floating point parameters. This is not desired behavior. It would eliminate the possibility of using the program's framework with other types problems.

The function dispatcher is used to disseminate information about the objective function being used. Through an inquiry to the function dispatcher, the number of parameters in the current objective function is stored in the variable NumberOfParameters. Again, the function dispatcher could be queried whenever the number of parameters is needed, but speed considerations necessitated querying only once.

Each of the parameters in a real vector objective function must be bounded. The two arrays RealMinimums and RealRanges store the bounds on each of the parameters. There are as many elements in each of these arrays as there are parameters required by the objective function. The values of these arrays are set through the function dispatcher. When the first RealIndividual is created, the constructor queries the function dispatcher for pointers to these arrays.

The mutation operation for a RealIndividual needs one controlling parameter: MutationRate. This value is controlled by an iterator and thus ultimately by the user. The Iterator pointed to by the variable RealIndividualIT controls this value.

Further definition of mutation is left out of this class. It is up to derived classes to define the exact interpretation of MutationRate and act on the genetic material. This allows several types of individuals to be created that work with different methods of mutation.

Crossover is also left undefined by this class. The classes derived from this one are free to define crossover in what ever manner they see fit.

4.2.1.2. RealIndividual0

The RealIndividual0 class derives from RealParameterIndividual and is the complete definition of the RVGA. It makes definitions for Mutation and Crossover that and defines the auxiliary variables needed to implement them.

```
class RealIndividual0 : public RealParameterIndividual
{
  static double *MutationRange;
  static Iterator *RealIndividual0IT;
  static double MutationSize;
  static int Usage;

 public:
  ...

  virtual Individual* Clone (void) { return new RealIndividual0(); }
  virtual void CrossOver (Individual*);
  virtual void Mutate (void);
};
```

Figure 4.3

The mutation rate from the parent class RealParameterIndividual is interpreted as the probability that a given parameter of the real vector will be mutated by the mutation routine. When the Mutate function is invoked for an individual, the first act is to decide if the mutation is to be effective. This is done by generating a random number between zero and one and comparing it with the mutation rate. If the random number is less than or equal to the mutation rate, the mutation proceeds.

Mutation consists of perturbing the value of parameter in the vector of parameters of an individual. The MutationSize variable places a bound on the size of the perturbation. For example, a value of .2 would allow a parameter to be changed up or down by ten percent of the total range of the parameter. The mutation size is controlled by an iterator.

The actual process of mutation first selects the direction of the mutation; either positive or negative. Second, a range of possible new values of the parameter is calculated. The range is adjusted to remain within the bounds of the parameter. Then a new value of the parameter is selected uniformly from the range.

For example, consider the value of a parameter to be 89 and the bounds of the parameter give it a possible range of 0 to 100. A mutation size of .25 would represent a mutation range of plus or minus 12.5. Because 89 plus 12.5 is greater than 100 (the upper bound on the parameter) the mutation range is adjusted to be minus 12.5 to plus 11. If the mutation were chosen to be negative, the new value for the parameter would be chosen from the range of 76.5 to 89. If the mutation were chosen as positive, the value would be in the range of 89 to 100.

The variable MutationRange is an array used to store the maximum size of the mutation range for each parameter. This array was added for speed considerations. It is best to calculate this only once and then lookup the value rather than calculating it each time the value is needed. The value of this array is calculated in the second constructor.

Crossover exchanges genetic material between two individuals. For RealIndividuals this is simply an exchange of a string of parameters. Selection of the string is done by first selecting a starting and an ending point. These can be thought of as the cut points. The parameters between the two cut points are then exchanged between the two individuals. If one individual has a parameter list [12.4, 16.8, 3.1, 8.9, 6.6] and the other has [11.4, 15.3, 5.6, 8.8, 4.5], crossover cut points at 1 and 3 would result in the parameters lists being: [12.4 15.3 5.6, 8.9 6.6] and [11.4, 16.8, 3.1, 8.8, 4.5].

4.2.1.3. BinaryIndividuals

BinaryIndividuals were created to mimic the behavior of the individuals in the GENESIS program exactly. The implementation of algorithms used in GENESIS are similar but in most cases are not exact matches to the implementations used here. Where the differences are significant, it will be noted.

BinaryIndividuals are derived from the intermediate class RealParameterIndividual. BinaryIndividual adds a new representation of the genetic material to the vector of floating point numbers provided by RealParameterIndividual. The new representation is the target of the mutation and crossover operations. When it is time to evaluate a BinaryIndividual, the representation is translated and stored in the vector of floating point numbers from RealParameterIndividual. This gives BinaryIndividuals the flexibility of using its own genetic material representation while still be able to exploit the features inherited from RealParameterIndividual.

```
class BinaryIndividual : public RealIndividual
{
  static int ListLength;
  static int NumberOfBytes;
  static int BitsPerParameter;
  static int MaxValueOfBitPattern;
  static int MuNext;
  static int UseGraycode;
```

```
            static Iterator *BinaryIT;

            static int Usage;

            unsigned char *List;

            public:
            ...
            void Decode (void);
            void PutItem (unsigned char, int);
            unsigned char GetItem (int);
            ...
        }
```

Figure 4.4

The representation of the genetic material of a BinaryIndividual is a string of bits. The bit string is referred to by the pointer List. The routines PutItem and GetItem allow for easy access to the individual bits within the string as if the list were an array of bits. List is the only instance variable added by this class.

ListLength holds the number of bits that are used as the genetic material. The NumberOfBytes variable stores the size in bytes of the list. Because the number of bits used is not necessarily evenly divisible by eight, there will be several unused bits at the end of the list. The BitsPerParameter variable stores the number of bits that are allocated for each parameter of the objective function. This value is controlled by an iterator and therefore by the user.

The bit pattern is converted into the floating point values used by the objective function. The bit pattern can be interpreted in one of two ways: either as a binary number or a gray coded binary number. The variable UseGraycode determines how the bit pattern is translated. UseGraycode is indirectly controlled by an iterator.

The bit pattern is translated into an integer that is then scaled into a range that is appropriate for the parameter that the bits represent. The process of scaling is assisted by the variable MaxValueOfBitPattern.

Mutation for a BinaryIndividual is done by inverting the value of a specific bit in the bit string. BinaryIndividual uses the mutation rate from RealIndividual to determine if a mutation occurs. However, the value is reinterpreted to mean the probability that a given bit is mutated rather than a given parameter is mutated.

One way of implementing mutation for BinaryIndividuals, would be to cycle through all bits, generating a random number for each and comparing it with the mutation rate. This technique, while effective, would be very slow. Instead of cycling through all bits, the number of bits between mutations can be calculated. MuNext is the variable that holds this interval. As each individual in a population is considered, this value is decremented according to the number of bits in an individual. As soon as the value of MuNext is less than the number of bits in an individual, MuNext is used to calculate an index to the bit that is to be mutated. The mutation occurs and then a new MuNext is calculated based on the mutation rate.

Like mutation, crossover operates on the bit string. Two individuals are chosen and two cut points are selected within the bit strings. The portion between the cut points is swapped. The cut points are not limited to the boundaries between parameters. Great care is taken during the execution of the crossover function to avoid unnecessary work. If the regions inside or outside of the cut points of the two individuals are identical, the crossover would result in no change to the individuals. If this condition is found to be true, the crossover does not take place.

BinaryIndividuals have an additional step added to the process of evaluation. It is necessary to translate the bit string into a vector of floating point numbers. The Decode routine accomplishes that task. Immediately before the objective function is applied to the individual in the CalcPerformance routine, the Decode routine is invoked.

4.2.1.4. RealIndividual1

RealIndividual1 is another class based on RealParameterIndividual. It defines its own versions of crossover and mutation in a different manner than RealIndividual0. Its definitions are created to simulate the mutate and crossover functions of BinaryIndividuals. RealIndividual1 does not include its own definition of the genetic material in the manner of the class BinaryIndividual, it uses the array of floating point numbers provided by the base class RealParameterIndividual.

```
class RealIndividual1 : public RealParameterIndividual
{
  static Iterator *RealIndividual1IT;
  static int VirtualBitsPerParameter;
  static int MaxValueOfBitPattern;
  static int Usage;

  void MutateParameter (int);

  public:
  ...

  virtual Individual* Clone (void) { return new RealIndividual1(); }
  virtual void CrossOver (Individual*);
  virtual void Mutate (void);
};
```
Figure 4.5

Mutation in RealIndividual1 is designed to simulate the mutation technique found in the class BinaryIndividual without using the BinaryIndividual genetic material representation. To facilitate this, a class variable called

VirtualBitsPerParameter is defined. This value is the equivalent of the BitsPerParamater variable found in BinaryIndividual and serves a very similar purpose. Just as the size of a given mutation is directly related to the number of bits per parameter in a BinaryIndividual, the size of a RealIndividual1 mutation is dependent on VirtualBitsPerParameter.

When a BinaryIndividual performs a mutation, it simply flips the value of one of the bits in its genetic material. This causes a perturbation in the value of the parameter from which the bit was selected. Consider a BinaryIndividual using a straight binary encoding for each parameter. If R_i is the range of a parameter x_i, then mutating bit number k causes a perturbation of the size $R_i 2^{-k}$. The sign of the perturbation depends on the original value of the bit. If the original value were 0, then the perturbation would be positive. A negative perturbation would result if the original value had been 1.[4, p7]

MutationRate is interpreted by RealIndividual1 in the same manner that it is interpreted by BinaryIndividual: the mutation rate reflects the probability that a given bit will be mutated. RealIndividual1 makes its decision to mutate on a parameter by parameter basis. Since the mutation rate specifies the probability that a bit is mutated and each parameter represents a collection of bits, the mutation routine compares a uniformly selected random number with the mutation rate multiplied by the number of virtual bits per parameter. If the random number is less than the product, a mutation occurs.

Simulation of binary mutation begins with the selection of virtual bit, k. This bit determines the maximum size of the perturbation. It calculates a maximum perturbation size with a formula that is the equivalent of $R_i 2^{-k}$ multiplied by 2. Then a new value for the parameter is chosen uniformly from the range of the current value to the current plus or minus the maximum size of the perturbation. The sign of the perturbation is chosen randomly. The multiplication by 2 is to adjust the average perturbation size to the same size as the mutation size in binary mutation. If adding or subtracting the maximum value of the perturbation results in a number that is out of range for the parameter, the maximum value of the perturbation is adjusted. This prevents the value of the parameter from stepping beyond its limits.

For example, a given parameter might have a range of 0.0 to 10.24 with ten virtual bits per parameter. If virtual bit number five is selected for mutation, this corresponds to a mutation range of plus or minus 0.32. The actual mutation proceeds from this point in the same manner as the mutation from the class RealIndividual0.

Simulation of crossover of BinaryIndividuals is a two stage process. The first stage exchanges parameters between two individuals using two crossover points. The second stage perturbs one of the parameters at each crossover point. This is in line with the effects of crossover on a BinaryIndividual. In the case of a BinaryIndividual, all the bits between the crossover points are exchanged between the two individuals. This could result in an exchange of the entire representation of several parameters. A crossover point can fall within the bit pattern of a parameter of a BinaryIndividual. This results in an exchange of bits between the parameters of the two individuals involved. In effect, this is a perturbation of the parameters.[4, p5]

In the beginning of the first stage, two crossover points are selected. These points are selected to cross between the parameters. The parameters between the crossover points are swapped in the same manner that crossover takes place for class RealIndividual0. One special case exists for the real parameter crossing: the two crossover points can be equal. While this is not allowed to happen in the case of crossover for the class RealIndividual0, RealIndividual1 uses this occurrence to simulate both crossover points in the same parameter. When the two crossover points are the same, no exchange of parameters happens. The crossover routine proceeds directly to the second stage.

The second stage of crossover perturbs a parameter immediately adjacent to each crossover point. It accomplishes this using a method very similar to mutation. However, the technique is a little more complex, because it must simulate the exchange of several bits between the parameters of two individuals.

Examining the effects of the perturbation of binary crossover in more detail will make the reasoning behind the simulation techniques clearer. Consider the effects of crossover for the two individuals below:

```
1 0 1 0 1 1 0 0 1 1
1 1 0 1 1 0 1 1 0 0
```

The bit strings are arranged with the bits numbered from zero to nine from right to left. The least significant bit is numbered with the zero. The crossover points are also numbered from zero to nine, with crossover point zero immediately to the right of bit zero. If a crossover point falls at position two, the BinaryIndividuals will exchange their two least significant bits. In this example, the first individual has 11 in the least significant positions, however it could have any of the four possible values for those two bits. The same is true for the other individual. At most, the new value that an individual receives for its two bits will be a difference of three from the old value. If the original bit pattern was 11 and the new bit pattern is 00, the difference is three. Graphing the outcome of all possible combinations of original and new bit patterns yields figure 4.6.

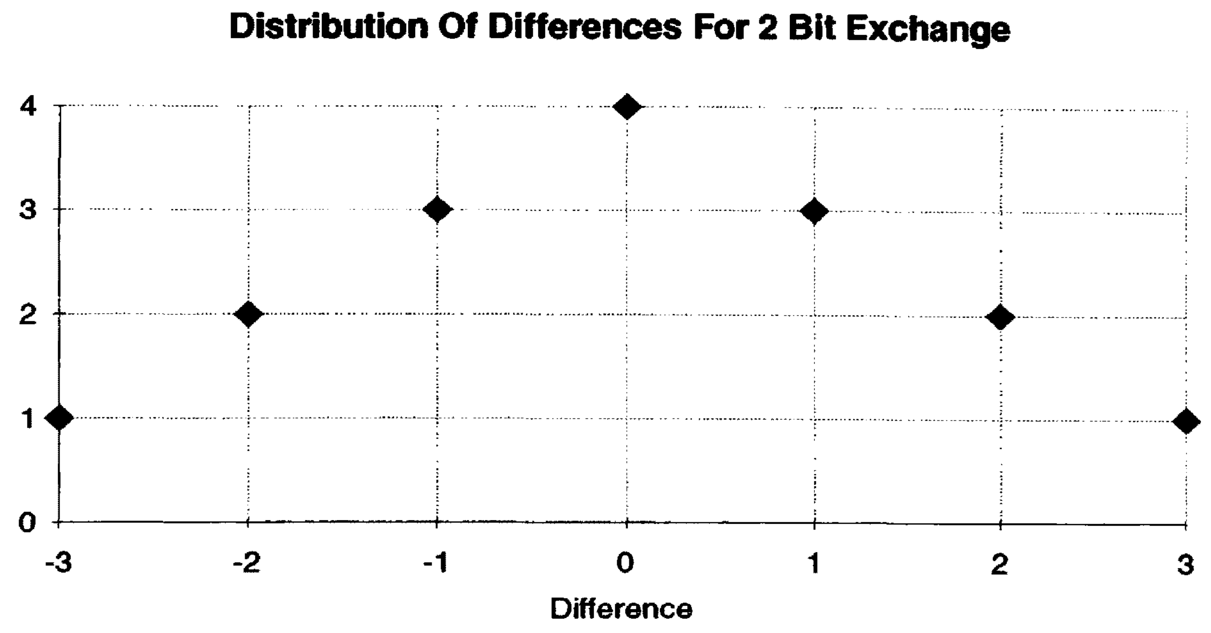

Figure 4.6

RealIndividual1 selects a value by which to perturb a parameter by selecting a perturbation from a distribution similar to the one in figure 4.6. The maximum size of the perturbation is determined by selecting the lesser of two values calculated using two different methods.

In the first method, RealIndividual1 selects a virtual crossover point based on the number of virtual bits per parameter. This is very similar to the method used to select the mutation size in the Mutate routine. This value becomes the first method's candidate for the maximum perturbation size.

When a population of binary individuals is approaching convergence, they begin to look very similar. In other words, as the individuals begin to cluster around the same values of their parameters, the binary encodings of the parameters become more and more alike. The probability that the exchange of bits in a binary crossover results in a perturbation begins to drop. To take this situation into account, the second method of determining a perturbation size simply calculates the difference between the two parameters from the two individuals. It uses this value as its candidate for the maximum perturbation.

The lesser of the two values calculated by the two methods is selected as the maximum perturbation size. The actual perturbation is chosen from the distribution given in figure 4.7 scaled into a range appropriate for the parameter that is to be perturbed.

The probability density function found in figure 4.7 represents a continuous version of the right half of the graph in figure 4.6. It shares the same distribution of values as the absolute value of the bit differences. The value zero occurs four times as often as the value three in both graphs.

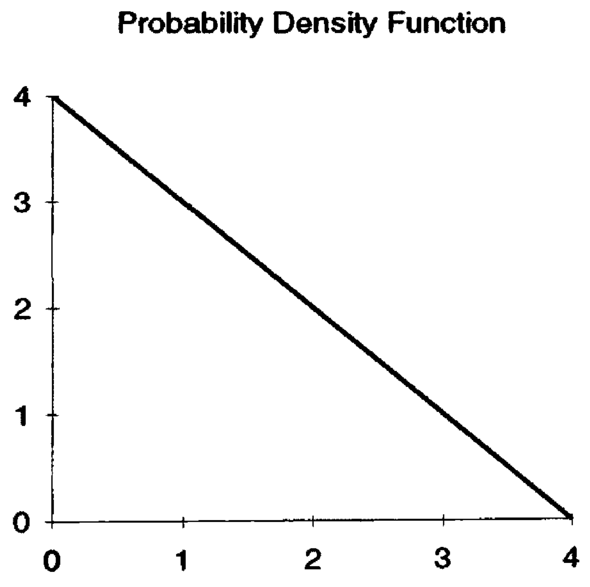

Figure 4.7

A random number can be selected from the distribution by first selecting a random number r uniformly from the range of 0.0 to 1.0. The number is then transformed by the equation: 1.0 - sqrt (1.0 - r). This number can then be scaled into the appropriate range and the sign selected randomly. The function of figure 4.8 is a routine that will select a random number from the right half of the distribution. When this routine is used to generate a random number from the distribution to be used as a perturbation, it is necessary to calculate the sign of the result separately.

```
double RandRangeLinear (double low, double high)
{
  return ((1.0 - sqrt (1.0 - Random())) * (high - low)) + low;
}
```

Figure 4.8

There is one further complication before the perturbation can be applied to the parameter. Since parameters are bounded, a facility must be created that will prevent them from over stepping their limits. In the case of mutation, this is done by restricting the size of the range from which the new value is selected. Since mutation uses a uniform distribution for selection, the range restriction has no side effects. With the perturbation selection, the distribution is not uniform. Simply restricting the range from which the value is selected skews the distribution. Figure 4.9 shows the problems with restricting the range of the maximum perturbation size. The figure shows that for a given parameter, the maximum perturbation size is four. However, the current value is such that adding a value of four would put the parameter over its upper bound. In fact, the perturbation must be selected to be one or less to prevent an out of range problem. The line composed of dashes represents the distribution if the perturbation range is simply limited to a size of one. The effective distribution shows that it is very difficult for the parameter to ever reach its upper limit.

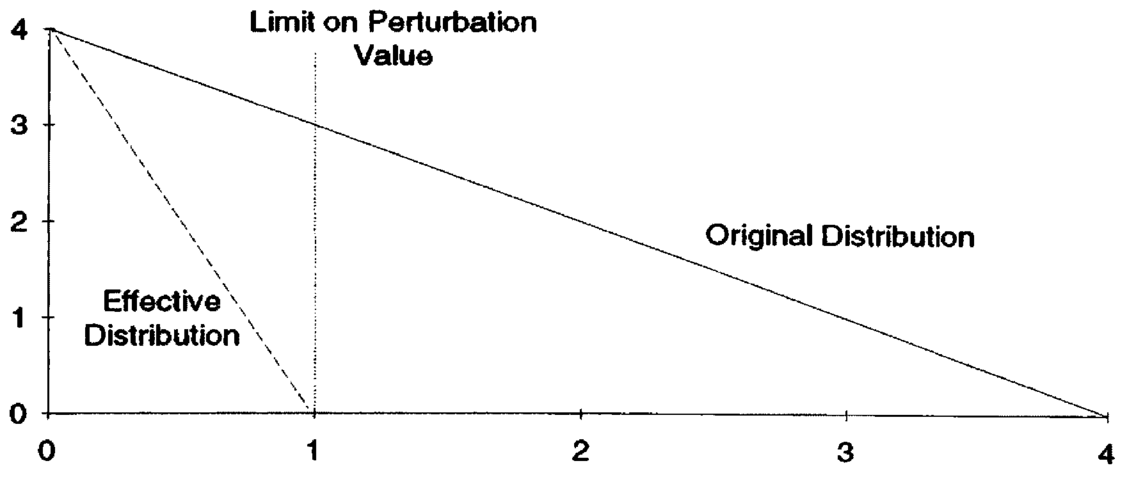

Figure 4.9

One possible solution to this problem is illustrated in figure 4.10. Here, the original distribution from the right of the limit is translated, scaled and then added to the original distribution on the left of the limit. This produces the distribution shown by the line composed of dashes. While this solution does skew the original distribution, it does not suffer from preventing the original parameter from reaching its bounds.

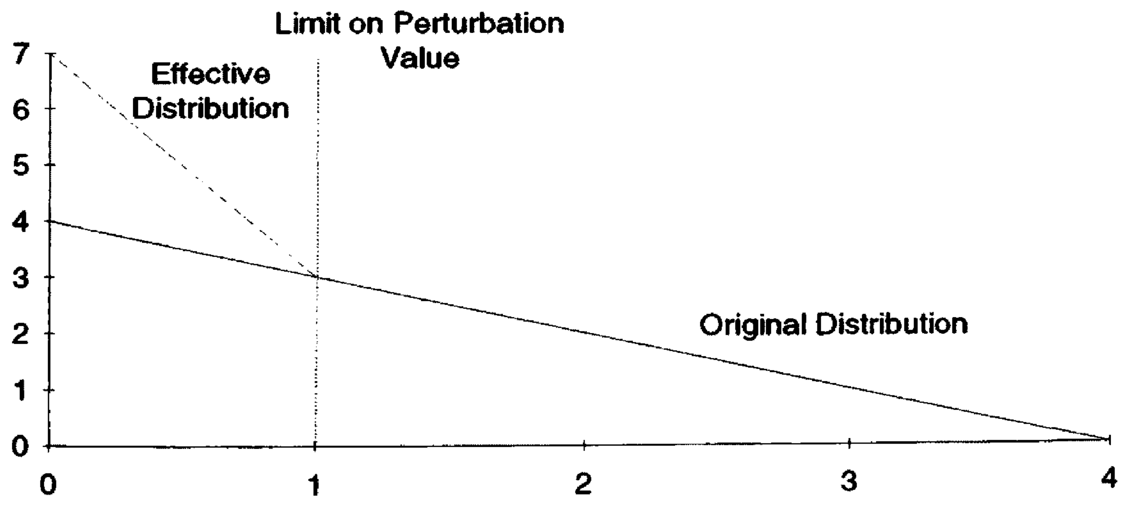

Figure 4.10

This solution requires a modification to the routine of figure 4.8 that selects a random number. A threshold value corresponding to the limit of the parameter is specified. The threshold partitions the range of the uniformly selected random variable, r, into two sections. Values below the threshold are used without modification. Values above the threshold are translated to below the threshold. The value is then scaled by a factor that is directly related to the relative sizes of the ranges above and below the threshold.

For example, the goal might be to select a perturbation from a range where the minimum is 0.0 and the maximum is 4.0. But the parameter that is to be perturbed is 1.0 away from its upper bound. This means that the perturbation must be selected from a range of 0.0 to 1.0.

The first step is to chose a uniform random number, r, between 0.0 and 1.0. This range represents then entire 0.0 to 4.0 range of the perturbation. Since the limit on the perturbation is 25 percent of the original maximum perturbation size, a threshold value of 0.25 is chosen. Any value of r above the threshold of 0.25 will be modified. If the selected number were .085, it would be translated by subtracting threshold to a value of 0.25, giving a value of 0.6. Since the proportion of the ranges above and below the threshold is 1/3, the value is scaled by dividing it by 3. This yields a value of .2. This value is then used as the value r. This new value can then be applied to the equation: 1.0 - sqrt (1.0 - r), yielding a normalized perturbation. This value is then scaled back to the original range of 0.0 to 4.0 given a real perturbation size of 0.8. The routine of figure 4.11 implements this technique.

```
double RandRangeLinearThreshold (double low, double high, double threshold)
{
  double r = (1.0 - sqrt ( 1.0 - Random()));
  if (threshold < high)
  {
    double NormalizedThreshold = (threshold - low) / (high - low);
    if (r > NormalizedThreshold) r = (r - NormalizedThreshold) *
                                     NormalizedThreshold /
                                     (1.0 - NormalizedThreshold);
  }
  return (r * (high - low)) + low;
}
```

Figure 4.11

4.2.2. Population

Population is a base class which serves as a starting place for the construction of populations. The class declares all of the features that a population must have while declining to make specific requirements about implementation details. The class Population is show in figure 4.12.

Just like the class Individual, Population has two constructors. The first constructor is used to create a prototype population. The second constructor is used in the process of cloning the prototype.

```
class Population
{
  static IteratorSet* IS;

  int PopulationSize;
  double AveragePerformance;
  Individual **List;

  int EvaluationMadeFlag;
  double CurrentPerformance;

  protected:
  static ProbeList Probes[10];
  Iterator* PopulationIT[2];
  Individual* WorstMember;
  Individual* BestMember;
  ...
```

```
        public:
        ...
        virtual Population* Clone (void) = 0;
        virtual int  AdvanceOneGeneration (Population*) = 0;

        virtual int MeasurePerformance (void);
        virtual void Print (FILE*);
        ...
    }
```

Figure 4.12

The PopulationSize variable is the only part of the base class Population that is under the control of an iterator. The variable PopulationIT points to the iterator. PopulationSize controls the size of the list of individuals in the population.

List is the variable that maintains the list of individuals. Actually, List points to an array of pointers to individuals. The array is dynamically allocated according to the PopulationSize variable. The array is of pointers to individuals rather than individuals, for two reasons. First, this allows the individuals to be of any type derived from the base class Individual. If an array of Individuals were used, Population would have to be recoded for every variant of individual created. Second, the array of pointers allows the list to be sorted or manipulated in a more efficient manner. Sorting requires only that the pointers be moved. If pointers were not used, it would be necessary to copy the individuals in order to move them. Pointers are significantly faster to manipulate.

Population maintains some basic information about the individuals it holds. AveragePerformance is calculated over the entire population whenever the population is evaluated. In addition, the population maintains two variables pointing to worst and the best member individuals.

Like the class Individual, Population does not stand alone. It requires that a derivative class be created to fill in details of the implementation. It specifies a minimum interface that the derived class must define. Included in the minimal interface are two routines: AdvanceOneGeneration and Clone. There are also two functions that a derived population type may redefine: MeasurePerformance and Print.

AdvanceOneGeneration is required to be defined by the derived population type. This routine defines the technique that moves a population from one generation to the next. It should outline each step of the process applying them to the individuals in the population.

Clone is a method of creating a population. When invoked, it should return a pointer to a new population of the same type as the current population. It is not necessary for the new population to be fully populated with individuals.

MeasurePerformance is fully defined by class Population. It cycles through the population applying each individual's CalcPerformance function. During this process, the population maintains its informational variables AveragePerformance, BestIndividual and WorstIndividual. A derived population may redefine this function as it sees fit. However, it is probably not necessary to do so.

There are two variables, CurrentPerformance and EvaluationMade which contain information regarding the status of the evaluation process during the execution of MeasurePerformance. CurrentPerformance is maintained with the value of the performance of the individual most recently evaluated. EvaluationMade is a flag states if the current individual being measured was applied to the objective function. If an individual survived crossover and mutation with no changes, then it does not need to be reapplied to the objective function. The old value of performance is still up to date. These two variables are created as a convenience for the probes Online and Offline that measure the on going performance of the algorithm.

The Print routine is also fully defined by the class Population. It simply cycles through the list of individuals invoking the Print function of each. In addition, it reports on the current value of AveragePerformance and the best and worst individual pointers.

4.2.2.1. GenesisPopulation

GenesisPopulation is a class derived from the base class Population. It is designed to conform with the performance of the GENESIS program. The functionality of populations in GENESIS have been translated into the framework of the base class Population.

```
class GenesisPopulation : public Population
{
  static double *WorstWindow;
  static double Worst;
  static int WorstWindowSize;
  static int WorstWindowPosition;
  static double GapSize;
  static int *Sample;
  static int *Sample2;
  static Iterator* GenesisPopulationIT[4];
  static int UseElitist;
  static double CrossoverRate;
  static int LastCrossoverIndex;
  static Usage;

  protected:
  void Gap (void);
  void GenerateNewWorst (void);
  void ClearWorstWindow (void);
  double AdjustWorstPerformance (double Adjustee);
  inline int GetLastCrossoverIndex (void);
  inline double GetCrossoverRate (void);
```

```
        public:
        ...
        virtual int AdvanceOneGeneration(Population*);
        virtual Population* Clone (void);
        virtual void Crossover (void);
        virtual void Mutate (void);
        virtual void Select (Population*);

        ...
    };
```

Figure 4.13

Many of the variables that were implemented as global variables in GENESIS have been placed as class variables in this class. This technique holds these variables in closer association to the code that uses them. Encapsulation in this manner, is an important feature of object oriented languages. Rather than discussing each variable individually, the discussion will be included with the routines that use the variables.

The AdvanceOneGeneration routine orchestrates the actions of a population evolving from one generation to the next. It is given a pointer to the population from the previous generation. It invokes the Select routine with that population. This creates the list of individuals that will become the new generation. The list of individuals is then associated with the population from which the AdvanceOneGeneration routine was invoked. It then executes the Mutate and Crossover routines on the list of new individuals. If the elitist strategy is enabled and if the best member from the parent (previous) generation is not present in the new list of individuals, it is added. This imitates the elitist strategy found in GENESIS. AdvanceOneGeneration then evaluates its list of individuals by invoking the MeasurePerformance routine.

The Select routine from GENESIS uses an technique called the Baker Selection Algorithm. Space in the new population is allocated to individuals based on their relative ranking. A given individual may be represented in the subsequent generation several times if its performance is sufficiently above that of its peers. The new population is assembled in the Select routine by first filling an array called Sample with the indices of individuals within the parent population. For example, if the Select routine determines that the first individual from the parent population is to be represented four times in the new generation, the index 1 will be placed in the first four positions of the Sample array. If the second individual from the parent population were to have a poor performance relative to the other members of the population, it might not survive into the next generation. The number 2 would not be placed in the Sample array. Generally if the normalized fitness of an individual is twice that of another, it will be allocated twice the space in the new generation. Once the Sample array is filled, it is shuffled and used to copy individuals from the parent generation into the new population.

At the discretion of the user, a generation gap may come into play during the selection process. The generation gap specifies a minimum percentage of a population that is to survive into the succeeding generation without regard to the Baker Selection algorithm. Immediately after the array Sample is filled by the Select routine, the Gap routine modifies the array. It overwrites elements of the Sample array with indices selected randomly from the parent population using the array Sample2 as an intermediate array. It replaces a percentage of the Sample array specified by one minus the variable GapSize. GapSize is controlled by an iterator. If GapSize were to be specified as .6, at least forty percent of the parent population would survive into the new generation.

The Crossover routine applies the Crossover operation to pairs of individuals in the population. CrossoverRate, controlled by an iterator, specifies the percentage of the population that is subjected to this process. If the CrossoverRate were specified to be 0.6 and there were one hundred individuals in the population, the first sixty individuals would be crossed in pairs. The variable, LastCrossoverIndex, is calculated to be the index of the last individual to be subjected to crossover. In the above example, the value in LastCrossoverIndex would be sixty. This variable is calculated by the constructor of the GenesisPopulation. It simplifies the calculation in the loop that applies the crossover to the individuals.

The Clone routine is a straight forward implementation of the requirements of the base class Population. Clone simply creates and returns a new instance of the class GenesisPopulation.

The Mutation routine is also very straightforward. It cycles through the list of individuals, applying the Mutation operation on each.

4.2.3. Genetic

The class Genetic is the supervisor for controlling the process of the genetic algorithm. It is responsible for the creation and disposal of populations. It orchestrates experiments by tracking each generation. It makes the decisions that result in the termination of experiments. Experiments are clustered in experiment sets. The Genetic class directs the advancement from one experiment to the next in an experiment set.

```
class Genetic
{
  ...

  long InitialSeed;
  int ExperimentsPerSetting;
  int MaxGenerations;
  int MaxTrials;
```

```
        int MaxSpin;

        int GenerationNumber;
        int Trials;

        Individual* PrototypeIndividual;
        Population* PrototypePopulation;

        Population* Parents;
        Population* Children;

        ...

        public:
        ...
        void GeneticAlgorithm (void);
        void RunExperimentSet (void);
        void Evolve (void);
        ...
    }
```

Figure 4.14

A single experiment is executed by the Genetic class with its Evolve routine. This routine sets up the initial population and then guides it through the process of a genetic algorithm. During the process, the Genetic class monitors the population while waiting to detect the experiment stopping criteria. As soon as it senses the stopping criteria, the Genetic class closes down the experiment in an orderly manner.

The Genetic class uses three techniques for stopping an experiment. These are maximum number of generations, maximum number of trials and maximum spin.

The MaxGenerations variable is controlled by an iterator. The Genetic class never lets an experiment continue past this ceiling on the number of generations.

A trial is defined as the application of the objective function on an individual that has not been tested in the past. The Genetic class counts the number of trials in each generation. If the number of trials exceeds the MaxTrials threshold, the experiment is terminated. The MaxTrials variable is controlled by an iterator.

MaxSpin is related to both the number of generations and the number of trials. A spin is counted if a generation passes with no trials occurring. The MaxSpin variable puts an upper limit on the number of generations that can pass without a trial. MaxSpin is also controlled by an iterator.

Before the beginning of an experiment, a sample population is created. This sample population is called the PrototypePopulation. This population is never used in the actual processing of the genetic algorithm. It purpose is to spawn new populations of its type at the beginning of each experiment. In the OBJGEN program, the PrototypePopulation will be a GenesisPopulation. However, using the PrototypePopulation scheme, this does not always have to be true. If a new type of Population is derived, it can be passed in as the prototype. Genetic class will then create populations of the new type without any recoding necessary.

```
Parents = PrototypePopulation->Clone();
Parents->Populate (PrototypeIndividual);
Parents->Initialize ();
SaveRandomSeed ();
Children = PrototypePopulation->Clone();
Children->Populate (PrototypeIndividual);
Trials = Parents->MeasurePerformance();
```

Figure 4.15

A similar scheme is in place for Individuals. The Genetic class is given a PrototypeIndividual. This individual is never processed through the algorithm, it is simply cloned to fill out populations at the beginning of an experiment. Again, this allows new variants of Individuals to be created and used with no recoding of the Genetic class or Population needed.

The first responsibility of the Genetic class in its Evolve routine is the creation of the population. It actually uses two populations which alternate roles as the previous (parent) population and succeeding (children) population. The Population pointers, Parent and Children, each refer to one of the two populations. The Parent population is created first by cloning the PrototypePopulation. The Parent population is then filled with individuals by cloning the PrototypeIndividual. The newly created individuals are initialized to a random state by the Initialize routine.

The second population is created by cloning the PrototypePopulation again and then assigning the result to the Children pointer. The new population is then filled with individuals of the appropriate type by cloning the PrototypeIndividual. These new individuals are not initialized to a random state.

The final step in the initialization process is to evaluate the performance of the first population. This is done by invoking the MeasurePerformance function on the Parent population pointer. It returns the number of trials in the first generation; invariably this is equal to the size of the population. The population is then ready to begin the process of evolution.

```
for (GenerationNumber = 1; (GenerationNumber < MaxGenerations) &&
                           (Trials < MaxTrials)
                           && (GenerationSpin < MaxSpin);
                           GenerationNumber++)
{
  PreviousNumberOfTrials = Trials;

  Trials += Children->AdvanceOneGeneration (Parents);

  T = Parents;
  Parents = Children;
  Children = T;
```

```
        if (Trials == PreviousNumberOfTrials)
          GenerationSpin++;
        else
          GenerationSpin = 0;
      }
```

Figure 4.16

The loop following the initialization phase of the Evolve routine is the main processor of the genetic algorithm. With each iteration of the loop, another generation passes. The loop is controlled by the three stopping criteria discussed above.

The AdvanceOneGeneration routine applied to the Children population causes individuals to be selected from the Parents population. The selected individuals are copied into the Children population. The processes of advancing one generation (Mutation, Crossover, Elitist and MeasurePerformance in the case of a GenesisPopulation) then act on the Children population.

The next step swaps the population pointers. The newly completed Children population becomes the new Parent population. The old individuals in the old parent population will be superseded in the next invocation of AdvanceOneGeneration. In this manner, the two populations trade roles in being the parent and children.

Before the next iteration of the loop, the stopping criteria are checked. If one of the three is deemed to be true, the loop is terminated. The two populations are then deallocated and the Evolve routine is completed.

```
void Genetic::RunExperimentSet (void)
{
  ...
  for (int i = 0; i < ExperimentsPerSetting; i++)
  {
    Evolve ();
  }
  ...
}
```

Figure 4.17

The Evolve routine represents only one experiment. Multiple invocations of the Evolve routine constitute an experiment set. The RunExperimentSet routine does exactly that. It is a simple loop that called the Evolve routine multiple times. The ExperimentsPerSetting variable puts an upper limit on the number of experiments that are run. ExperimentsPerSetting is controlled by an iterator.

```
void Genetic::GeneticAlgorithm (void)
{
  InitialSeed = ((LongIntIterator*) GeneticIT[8])->GetValue();

  do
  {
    MaxGenerations        = ((IntIterator*) GeneticIT[0])->GetValue();
    MaxTrials             = ((IntIterator*) GeneticIT[1])->GetValue();
    MaxSpin               = ((IntIterator*) GeneticIT[2])->GetValue();
    ExperimentsPerSetting = ((IntIterator*) GeneticIT[3])->GetValue();

    RunExperimentSet ();
  }
  while (IS->Step());

}
```

Figure 4.18

The GeneticAlgorithm routine of class Genetic controls all experiment sets. It also is responsible for controlling the IteratorSet which in turn controls all of the variables that have associated iterators.

The main loop of this routine runs experiment sets. It begins by retrieving values for its own variables from the iterators that control them. It then invokes the

RunExperimentSet routine. This will run a number of experiments depending on the value of ExperimentsPerSetting. All variables that have associated iterators will retain their values across an entire experiment set.

The loop is controlled by the IteratorSet IS. At the end of the loop, the Step function is called for the IteratorSet. This causes the iterators to advance to their next values. If all the iterators have exhausted their ranges of values, the Step routine returns a zero and the loop terminates. If the iterators have not exhausted their ranges, the loop continues. The variables controlled by iterators will pick up their new values and an experiment set will be run with those new values.

4.2.4. Probes

Probes are used extensively in the genetic algorithm and are responsible for a great deal of the program's power. The probes defined in the current version of the program span from very simple "variable dump" probes to complex probes that monitor other probes.

4.2.4.1. ProbeLists

ProbeLists are distributed in the Population, GenesisPopulation and Genetic classes in the program. This allows probes to monitor the progress of the genetic algorithm at most of the critical processing areas.

In the Genetic class there are five ProbeLists. They are associated with the following locations: before a set of experiments; after a set of experiments; before an experiment; after an experiment; and after the initial generation of an experiment is created. Assignment of a probe to one of these locations is done in a straight forward manner; a constant used to identify the location. The routine

AssignProbe is used to associate the measurement/function pair to the appropriate location.

Since GenesisPopulation is a derived class of Population, they share a set of ProbeLists and the assignment function. The ProbeLists are physically within the Population class, but only two of them are associated with code locations in the methods of class Population. The others are left for allocation by derived classes. GenesisPopulation uses five of the ProbeLists.

In Population, the two ProbeLists are used in the MeasurePerformance routine. One resides inside the inner most loop and is traversed after each Individual is measured for performance. The other is traversed after all of the measuring of Individuals is complete.

In GenesisPopulation, the lists are traversed after each stage of transforming a population from generation to the next. These are located in order at: after selection, after mutation, after crossover, after elitist and after the population is evaluated. This provides a good covering of possible locations for probes to work.

Figure 4.19 represents all of the existing ProbeLists throughout the OBJGEN program. The table shows the location of each list by class name and function name. The flag is the location code that is used when assigning a probe to a location. The flag also provides a terse description of the location.

Class	Function	Flag
Genetic	RunExperimentSet	GENETIC_PRE_EXPERIMENT_SET
Genetic	RunExperimentSet	GENETIC_POST_EXPERIMENT_SET
Genetic	Evolve	GENETIC_PRE_EXPERIMENT
Genetic	Evolve	GENETIC_POST_FIRST_GENERATION
Genetic	Evolve	GENETIC_POST_EXPERIMENT
Population	MeasurePerformance	POPULATION_IN_MEASUREPERFORMANCE
Population	MeasurePerformance	POPULATION_POST_MEASURE-PERFORMANCE
GenesisPopulation	AdvanceOneGeneration	GENESIS_POPULATION_POST_SELECT
GenesisPopulation	AdvanceOneGeneration	GENESIS_POPULATION_POST_MUTATE
GenesisPopulation	AdvanceOneGeneration	GENESIS_POST_CROSSOVER
GenesisPopulation	AdvanceOneGeneration	GENESIS_POST_ELITIST
GenesisPopulation	AdvanceOneGeneration	GENESIS_POPULATION_POST_MEASURE-PERFORMANCE

Figure 4.19

4.2.4.2. Population Dump Probe

The population Dump Probe is very simple probe. It collects no information. It periodically tells the population that it monitors to print itself. This is equivalent to the dump interval found in Genesis. The user selects the interval in units of generations and a destination for the report. The effect is to see each individual in a population every n (specified by the user) generations during the run of the program.

Dump defines the first (initialization) stage of the reporting stages and the last stage (reporting). It does not collect or manipulate any information, therefore it can ignore those stages.

The initialization stage simply resets its internal counter that tracks how many generations have passed. The Dump probe and its Reset function are assigned to the "before experiment" ProbeList of the Genetic class. This means that the probes internal counter will be reset at the beginning of each new experiment.

The report stage is assigned to the "after measurement" ProbeList of the Population class. After a population is evaluated, this probe will be invoked with its Report function. Since the function will receive a pointer to the Population as a parameter, the probe simply invokes the Population's print routine.

4.2.4.3. BestPopulation Probe

The BestPopulation class draws its lineage from both the Probe base class and the Population class. The objective of this class is to monitor each generation in an experiment and collect copies of the best individuals found. It gets the ability to hold a group of individuals from the class Population. The probe uses the methods from that class to manage its population of individuals. The user specifies how large the population is to be. As a probe, this class has the capability of browsing through a population to find the best individuals.

The first stage of reporting is implemented in BestPopulation's Reset function. Assigned to the "before experiment" ProbeList of the Genetic class, this function initializes the best population to the current type of individual. It accomplishes this through the use of the PrototypeIndividual. It gains access to the PrototypeIndividual through the pointer to the Genetic class instance that it receives as a parameter.

The Collect function is assigned to the "after measurement" ProbeList of the class Population. After all the individuals in a population have been evaluated, this probe compares the best individuals to the individuals it has already collected. If the best in the newly evaluated population is better than the worst of the collected population, the probe takes action. It copies the best individual it found over the worst it had collected previously.

The Report function is assigned to the "after experiment" ProbeList of the Genetic class. When an experiment ends, the Report function invokes the Print function from the probe's Population heritage. The effect is to have a list of the best individuals from an experiment printed at the end of each experiment.

4.2.4.4. AverageMeasurements Probe

The AverageMeasurements probe is an extendable probe. It maintains a list of other probes for which it calculates averages. In addition to inheriting the properties of a probe, it also inherits the capabilities of a linked list. This enables it to manage its list in a manner synchronized with its duties as probe.

The list of this probe consists of nodes called Averagers. This class inherits from the LinkNode class. The class Averager consists of the variables necessary for the calculation of an average and a standard deviation: a sum, a sum of the squares and counter of the number of elements to be averaged. In addition there is a pointer to a probe from which the data will be collected. The Collect routine of this class causes the Averager to get the current value of the probe it is monitoring. It uses the probe's GetProbeValue function. It takes the value it receives from the probe and adds it to its sum variable. It also tracks the sum of the squares of the values it receives. When requested through calling its GetAverage or GetStdDev it will make the appropriate calculations and return a value.

The AverageMeasurements probe will create and add to its list an Averager when it is given a pointer to a probe. It is, of course, important to give AverageMeasurements a probe of suitable type. A suitable type is a probe that collects floating point measurements.

The Reset function of AverageMeasurements traverses its list of Averagers invoking their Reset functions. The effect is to cause all Averagers to clear their sums and element counters. The Reset function is assigned to the "before experiment set" ProbeList of the Genetic class.

The Collect function traverses the list and induces the Averagers to poll their respective probes. Since the Collect function is assigned to the "after experiment" ProbeList, the MeasurementAverager class collects only the ending value of each probe. The consequence is to calculate an average of probe values over a set of experiments.

The Report function is assigned to the "after experiment set" ProbeList. When an set of experiments is completed, the MeasurementAverager reports the average and standard deviation of the probes it monitored. By traversing its list twice, printing the names of the probes it monitored in the first pass and printing the averages in the second pass, the MeasurementAverager prints a formatted report.

4.2.4.5. Online Probe

The Online evaluation of performance from the original Genesis program is implemented as a probe in OBJGEN. The Online measure of performance measures on going performance by generating an average of the performance of all new individuals. A new individual is defined as an individual that has been applied to the objective function. This the same concept as a trial. Whenever an individual is applied to the objective function a trial occurs. Online performance calculates an average of the performances from each trial.

Online's Reset function is assigned to the beginning of each experiment. The running total of performances as well as the counter of the number of trials are set to zero by this function.

The Collect function is assigned a location within the code of the MeasurePerformance routine of the class Population. MeasurePerformance cycles through the list of Individuals in the population and applies the objective function. Only individuals that are modified by Mutation or Crossover are subjected to the objective function. The ProbeList that Online's Collect function is assigned to is within the loop that cycles through the individuals. After each iteration of the loop, the probe list is traversed. If a trial has occurred, the Collect function will add the performance of the current individual to its running total and increment its trial counter.

Online's Report function is not assigned to a probe list. Online is assigned to the AverageMeasurements probe. The AverageMeasurements probe tracks the value of Online through the use of Online's GetProbeValue function. In this manner, Online abdicates its reporting responsibility and gives it to the AverageMeasurement probe.

4.2.4.6. Offline Probe

The Offline probe works almost identically to the Online probe. The intention of this probe is to measure convergence by generating an average of the best individuals. Whenever a trial occurs, the probe adds the best individual's performance to its running total and increments its trial counter.

For example, the sequence of trials might produce individuals with the following performances: 3.1, 2.8, 6.2, 5.4, 1.8 and 2.3. Since a better performance is considered to be a lower number, the offline measure will add the following sequence to its running total: 3.1, 2.8, 2.8, 2.8, 1.8 and 1.8. At each step, only the best performance seen so far will be added to the running total.

The distribution of Offline's functions are the same as Online's. The Reset function is placed on a probe list at the beginning of each experiment. The Collect function is on the probe list in the loop of the MeasurePerformance routine of the class Population. The Report function is not assigned. AverageMeasurements takes over the reporting function.

4.2.5. Iterators

Iterators are used extensively in the OBJGEN program. They provide the user with control over the parameters for the genetic algorithm. All incoming communication from the user that is intended to direct the controlling parameters of the genetic algorithm is fielded by iterators. Because Iterators provide a simple and consistent interface for the user, they are used even where multiple values for parameters are not allowed.

In the family of classes that comprise individuals, there are two constructors. The first constructor is used to create the prototype individual. This individual is created at the beginning of the program and is used there after to spawn all other individuals. These first constructors allocate iterators to control the variable parameters of individuals. Examples include the objective function number from the base class, and the number of bits per parameter from the derived class BinaryIndividual.

When an individual is created, all of the constructors traced back to the base class are executed. If a BinaryIndividual is created, the constructor for class

Individual is executed first. When that constructor completes, the next constructor down the inheritance chain begins. This means that the RealParameterIndividual constructor is next. Finally the BinaryIndividual constructor executes.

This implies that iterators allocated by the base class constructor are created first. Therefore, the user is prompted for values for these iterators first. The iterators for RealParameterIndividual are next, followed immediately by the iterators for BinaryIndividual. This ordering is also important for determining the nesting levels of the parameters. The first iterator allocated is the outermost in terms of nesting.

Once the prototype individual has been created, the first of the two constructors for each class is retired. They will not be used again during the execution of the program. Later, when filling out populations with individuals, the second constructors are used. Rather than allocating new iterators, these constructors use the iterators that already exist. The constructors query the iterators for the values needed to create individuals. For example, the second constructor for the class BinaryIndividual needs to allocate memory for the binary string that is to be the genetic material for the individual. It queries the iterator that was allocated by the first constructor for the bits per parameter value. The iterator returns the current value for bits per parameter and the second constructor then uses it in a calculation to determine how much memory to allocate.

The family of classes for populations follows the same scheme of having two constructors. The first constructor allocates iterators and the second constructor uses them. Examples of parameters controlled by iterators in populations include: the population size and the crossover rate.

The class Genetic allocates several iterators in its constructor. It does not have a second constructor. The iterators are queried for their values during the processing of other functions of the class.

Genetic controls has control of the program's only IteratorSet. The nesting of loops defined by iterator set is executed in the GeneticAlgorithm function of this class. See the code fragment on page 78 for a listing of GeneticAlgorithm.

The following table outlines the use of iterators within the code of OBJGEN. The first column indicates the classes that use iterators. The second column reveals the functions within the classes that allocate iterators. The third column indicates the function from which the iterators' GetValue function is called. The name of the parameters controlled by iterators is in the fourth column. The type of the parameter is given in the fifth column. Finally, the sixth column indicates if the user is allowed to supply multiple values for the parameter.

Class	Allocating Function	Used in Function	Parameter Controlled	Type	Multiple Values
Individual	Constructor #1	Constructor #2	FunctionNumber	int	yes
RealParameter-Individual	Constructor #1	Constructor #2	MutationRate	double	yes
RealIndividual0	Constructor #1	Constructor #2	MutationSize	double	yes
RealIndividual1	Constructor #1	Constructor #2	VirtualBitsPer-Parameter	int	yes
BinaryIndividual	Constructor #1	Constructor #2	NumberOfBits	int	yes
BinaryIndividual	Constructor #1	Constructor #2	UseGraycode	string	yes
Population	Constructor #1	Constructor #2	PopulationSize	int	yes
Genesis-Population	Constructor #1	Constructor #2	CrossoverRate	double	yes
Genesis-Population	Constructor #1	Constructor #2	WorstWindowSize	int	yes
Genesis-Population	Constructor #1	Constructor #2	GenerationGap	double	yes
Genesis-Population	Constructor #1	Constructor #2	UseElitist	string	yes
Genetic	Constructor	GeneticAlgorithm	MaxGenerations	int	yes
Genetic	Constructor	GeneticAlgorithm	MaxTrials	int	yes
Genetic	Constructor	GeneticAlgorithm	MaxSpin	int	yes
Genetic	Constructor	GeneticAlgorithm	InitialSeed	long	no
RealIndividual-Manager	Constructor	Constructor	Individual Type*	int	no
BestPopulation	Constructor	Constructor	Output File Name*	string	no
PopulationDump	Constructor	Constructor	DumpInterval	int	no
PopulationDump	Constructor	Constructor	Output File Name*	string	no
Average-Measurements	Constructor	Constructor	Output File Name*	string	no

* The parameters controlled by these iterators are used only once. There is no variable name with greater than local scope associated with these iterators.

Figure 4.20

4.2.6. Miscellaneous

There are several classes that serve in auxiliary roles in OBJGEN. These classes handle allocation chores for resources needed by the genetic algorithm. Included in the auxiliary classes are: FunctionDispatcher, OutputFileManager and Individual Manager.

4.2.6.1. FunctionDispatcher

Function dispatchers provide a method of handling a group of objective functions. When an individual needs an objective function, it can request one from the function dispatcher. Because of the huge variety of functions that might be needed by the various types of individuals, the class FunctionDispatcher defines very little. Derivatives of this class must be defined to handle all but the most basic of functions. In fact, FunctionDispatcher is so general in scope, it can handle only the names of functions. It cannot even handle the functions by itself.

This limitation is due to fact that the language C++ requires that pointers to functions be typed by their signature: the return type and parameter list. An individual designed for a combinatorial problem needs an entirely different sort of objective function than an individual designed for a parametric equation. These functions have different signatures and therefore require a different type of function pointer. Requiring derived types to define a pointer to the correct type of function is simpler than using a generic pointer type in the base class. The generic pointer would require casting to the correct type whenever the pointer was used.

```
class DoubleFunctionDispatcher : public FunctionDispatcher
{
  DoubleFunctionPointer *Functions;
  int *NumberOfParameters;
  double **Ranges;
  double **Minimums;

  public:
  DoubleFunctionDispatcher (int);

  DoubleFunctionPointer GetFunction (int n) { return Functions[n]; }
  void SetFunctions (DoubleFunctionPointer* fp) { Functions = fp; }

  int GetNumberOfParameters (int n) { return NumberOfParameters[n]; }
  void SetNumberOfParameters (int *np) { NumberOfParameters = np; }
  double* GetRanges (int n) { return Ranges[n]; }
  void SetRanges (double** r) { Ranges = r; }

  double* GetMinimums (int n) { return Minimums[n]; }
  void SetMinimums(double** m) { Minimums = m; }
};
```

Figure 4.21

The DoubleFunctionDispatcher is designed for managing a list of functions with real parameters. Initialization consists of providing an array of pointers to functions. In addition, DoubleFunctionDispatcher is capable of dispensing information about the functions it contains. This information consists of: the number of parameters required by the function, and the bounds on each parameter in the form of a minimum value and a range. Because DoubleFunctionDispatcher inherits from FunctionDispatcher, it is also able to return the name of the function.

```
char Function0_Name[] = "Function0";
const int Function0_NumberOfParameters = 2;
double Function0_Ranges[] = { 10.0, 10.0};
double Function0_Minimums[] = { -5.0, -5.0 };
double Function0 (double* x)
{
  return x[0] * x[1];
}

char Function1_Name[] = "Function1";
const int Function1_NumberOfParameters = 4;
double Function1_Ranges[] = { 10.0, 10.0, 10.0, 10.0};
double Function1_Minimums[] = { 0.0, 0.0, -10.0, -10.0 };
double Function1 (double* x)
{
  return x[0] * x[2] + x[1] * x[3];
}

DoubleFunctionPointer Functions[] = { Function0, Function1 };
char *FunctionNames[] = { Function0_Name, Function1_Name };
int NumberOfFunctionParamters[] = { Function0_NumberOfParameters,
                                    Function1_NumberOfParameters };
double *FunctionRanges[] = { Function0_Ranges, Function1_Ranges };
double *FunctionMinimums[] = { Function0_Minimums, Function0_Minimums };
```

Figure 4.22

In this example, two functions and all of their auxiliary information are defined. The information is then collected into arrays. The arrays are accessed in parallel. The DoubleFunctionDispatcher is initialized using a series of functions that notify the manager of the existence of the arrays. These functions are: SetFunctions, SetNumberOfParameters, SetRanges and SetMinimums. The function names are assign to the DoubleFunctionDispatcher by the inherited routine SetNames.

Once the arrays have been assigned to the DoubleFunctionDispatcher, an individual may retrieve any element of one the arrays through the use of a function and an index. For example, if an individual wants to use the second function, it can get a pointer to the function by invoking the GetFunction method on the DoubleFunctionDispatcher. The individual must pass the number one with its request for the objective function. The functions GetNumberOfParameters, GetName, GetRanges and GetMinimums allow the individual to retrieve the auxiliary information from the DoubleFunctionDispatcher.

4.2.6.2. FileManager

The FileManager class was created to eliminate conflicts between Probes that need to write to the same output file. Every probe has the ability to write its report to an output file. It cannot be guaranteed that a probe will be constructed to cooperate with other probes in opening and closing files. Data would be lost if two or more probes open the same file independently and both attempt to write to the file.

One method of preventing conflicts is to require that a probe open its output file immediately before writing to it then requiring the probe to close the file immediately afterward. Since no parallel processing is available in this program, this technique would insure that no two probes have the same file open at the same time. Unfortunately, if a probe is writing its output often, this technique suffers from excessive overhead. Opening and closing files are not quick processes.

The FileManager solves the problem very easily. Rather than opening files themselves, probes request files to be opened by the FileManager. The FileManager will open the file and return a file pointer for the probe to use. If the file has been opened previously by another probe, the FileManager will return the file pointer that has already been created. This insures that there will be only one file pointer per output file. Two or more probes may have the same pointer, but this is not a conflict. Probes may write to their output file completely unaware that other probes are using the same file.

```
class OutputFileManager
{
  char *FileNames[MAX_OUTPUT_FILES];
  FILE *FilePointers[MAX_OUTPUT_FILES];
  int Index;

  public:
  OutputFileManager (void);
  ~OutputFileManager (void);
  FILE* OpenFile (char*);
};
```

Figure 4.23

The output file manager maintains two arrays. The first array is a list of the names of the open files. The second is a parallel array of the pointers that are associated with the file names. The routine OpenFile is given a name of a file to open. It first searches the list of names for the name it was given. If it finds the name, it returns the associated file pointer from the pointer list. If the name was not found, it copies the name into the first free space in the name list and then opens the file. When the destructor is called for the OutputFileManager, all of the output files are closed. The probes that use the file manager do not have be concerned about closing the files.

4.2.6.3. RealIndividualManager

The sole purpose of the IndividualManager is to allow the user of OBJGEN to select the prototype individual from several types of Individuals. The constructor simply lists the types available and allocates an iterator to prompt the user. The user selects the number corresponding to the individual type desired for the run of OBJGEN. The user is not allowed to type more than one response. The constructor then creates an individual based on the user's selection

The function GetPrototype returns a pointer to the individual it created. OBJGEN then continues, using IndividualManager's individual as the prototype individual.

4.2.7. Mainline Program

OBJGEN's mainline program does little more than declare instances of the major classes, provide links between them and then invoke their actions. It does not have to mange any output or work in any supervisory capacity.

The code fragment of figure 4.24, demonstrates how the IteratorSet is constructed. If an input file was provided on the command line, it is opened. If no input file was specified, the standard input (terminal) is selected for input. The IteratorSet requires only a source file for initialization.

```
FILE *Source = (argc > 1) ? fopen (argv[1], "r") : stdin;
...
if (!Source)
{
  printf ("Cannot find %s\n", argv[1]);
  exit (-1);
}
...
IteratorSet IS(Source);
```

Figure 4.24

The function dispatcher is set up by merely declaring its existence and size. The functions and information about them are assigned to the dispatcher in groups.

```
DoubleFunctionDispatcher DFD(11);
DFD.SetFunctionNames (FunctionNames);
DFD.SetFunctions (Functions);
DFD.SetNumberOfParameters (NumberOfFunctionParamters);
DFD.SetRanges (FunctionRanges);
DFD.SetMinimums (FunctionMinimums);
```

Figure 4.25

The OBJGEN program is set up to work with three types of Individuals: RealIndividual0, RealIndividual1 and BinaryIndividuals. The PrototypeIndividual must be set to be one of these types. The RealIndividualManager allows any of the types to be selected by the user. In this code fragment, the RealIndividualManager is declared and invoked. The GetPrototype routine queries the user for the type of individual desired and then returns an individual of the user's choice.

```
RealIndividualManager RIM (&IS, &DFD, Source);
Individual* Ind = RIM.GetPrototype();
```

Figure 4.26

The next step declares the PrototypePopulation that will be used. This program is dedicated to using a GenesisPopulation. Immediately afterward, the instance of the Genetic class is declared and given pointers to the IteratorSet and the population and individual prototypes.

```
GenesisPopulation GP (&IS);
Genetic GA (&IS, &GP, Ind);
```

Figure 4.27

The next step is to create the Probes that will be used in the program. Each one is declared and then its functions are assigned to locations within the code of the rest of the program.

```
BestPopulation BP(&IS);
BP.Populate (Ind);
GA.AssignProbe (&BP, PROBE_RESET, GENETIC_PRE_EXPERIMENT);
GA.AssignProbe (&BP, PROBE_REPORT, GENETIC_POST_EXPERIMENT);
GP.AssignProbe (&BP, PROBE_COLLECT, POPULATION_POST_MEASUREPERFORMANCE);

PopulationDump PD(&IS);
GA.AssignProbe (&PD, PROBE_RESET, GENETIC_PRE_EXPERIMENT);
GP.AssignProbe (&PD, PROBE_REPORT, POPULATION_POST_MEASUREPERFORMANCE);

OffLineMeasurement OFF(&BP);
GA.AssignProbe (&OFF, PROBE_RESET, GENETIC_PRE_EXPERIMENT);
GP.AssignProbe (&OFF, PROBE_COLLECT, POPULATION_IN_MEASUREPERFORMANCE);

OnLineMeasurement ON;
GA.AssignProbe (&ON, PROBE_RESET, GENETIC_PRE_EXPERIMENT);
GP.AssignProbe (&ON, PROBE_COLLECT, POPULATION_IN_MEASUREPERFORMANCE);

AverageMeasurements AM (&IS);
AM.AddMeasurement (&BP);
AM.AddMeasurement (&OFF);
AM.AddMeasurement (&ON);
GA.AssignProbe (&AM, PROBE_RESET, GENETIC_PRE_EXPERIMENT_SET);
GA.AssignProbe (&AM, PROBE_COLLECT, GENETIC_POST_EXPERIMENT);
GA.AssignProbe (&AM, PROBE_REPORT, GENETIC_POST_EXPERIMENT_SET);
```

Figure 4.28

The last step is to invoke the genetic algorithm. The routine GeneticAlgorithm starts the experiments, executes them, and then shuts them down. When this routine is complete, the program is finished.

```
        GA.GeneticAlgorithm ();
```
Figure 4.29

4.3. Comparing BVGA and RVGA

The classes derived from Individual, BinaryIndividual and RealIndividual1, are designed to give similar results while exploiting different techniques. BinaryIndividuals imitate the GENESIS program using the same methods employed in that program. The class RealIndividual1 simulates the class BinaryIndividual using a different data representation (see page 58). RealIndividual1, using only a floating point vector genetic material representation, should deliver results very similar to those given by BinaryIndividual using a bit string genetic material representation.

The OBJGEN program can be used to run experiments with either type of individual. The same objective functions can also be applied to either type. Since OBJGEN generates reports that are independent of the type of individual it uses, it is set up ideally for the comparison of multiple types of Individuals.

The experiments were set up with the first five of the classic deJong problems (F1 through F5) [5]. The sixth problem (F6) is deJong's problem number five (Shekel's Foxholes) rotated thirty degrees in the plane [4].

Figure 4.30 shows the input file used to run the program for BinaryIndividuals on five of the six problems. The second line shows the input that informed the program to run the functions numbered 1, 2, 3, 5, and 6. Each of these functions used all the combinations of the other parameters. Function number 4 was excluded from this list to allow for a different number of bits per parameter than the other functions. As the figure shows, the use of graycode and the crossover rate were the only two other settings that were given multiple values.

```
2                   ;Individual Type
1 2 3 5 6           ;Function Number +
0.01                ;Mutation Rate +
10                  ;Bits Per Parameter +
y n                 ;Use Graycode +
50                  ;Population Size +
0.0 0.8             ;Crossover Rate +
5                   ;Worst Window Size +
1.0                 ;Generation Gap +
y                   ;Use Elitist +
2000                ;Maximum Generations +
2000                ;Maximum Trials +
10                  ;Maximum Spin +
1000                ;Experiments Per Setting +
876543              ;Initial Seed
5                   ;Population Size +
nul                 ;Best Population Report Destination
0                   ;Population Dump Interval
stdout              ;Population Dump Destination
all_2_r.xls         ;Measurement Report Destination
```

Figure 4.30

A file, very similar to the one in figure 4.30, was constructed for the run of the program with RealIndividual1. Of course, the file for RealIndividual1 did not have an option for graycode. All the other parameters were run with the same settings found in figure 4.30.

The Offline measure of performance (see page 85) was used as the metric for comparison. In this measure, a lower value corresponds to "better" results.

The first experiment ran with no crossover. This isolated the mutation operation and provided the opportunity to examine the effectiveness of the RealIndividual1 mutation technique. In all cases, the RealIndividual1performed at least as well as the BinaryIndividual tests. With the exception of F5 and F6, RealIndividual1 performed similarly with BinaryIndividual. The chart in figure 4.31 shows the Offline performance data.

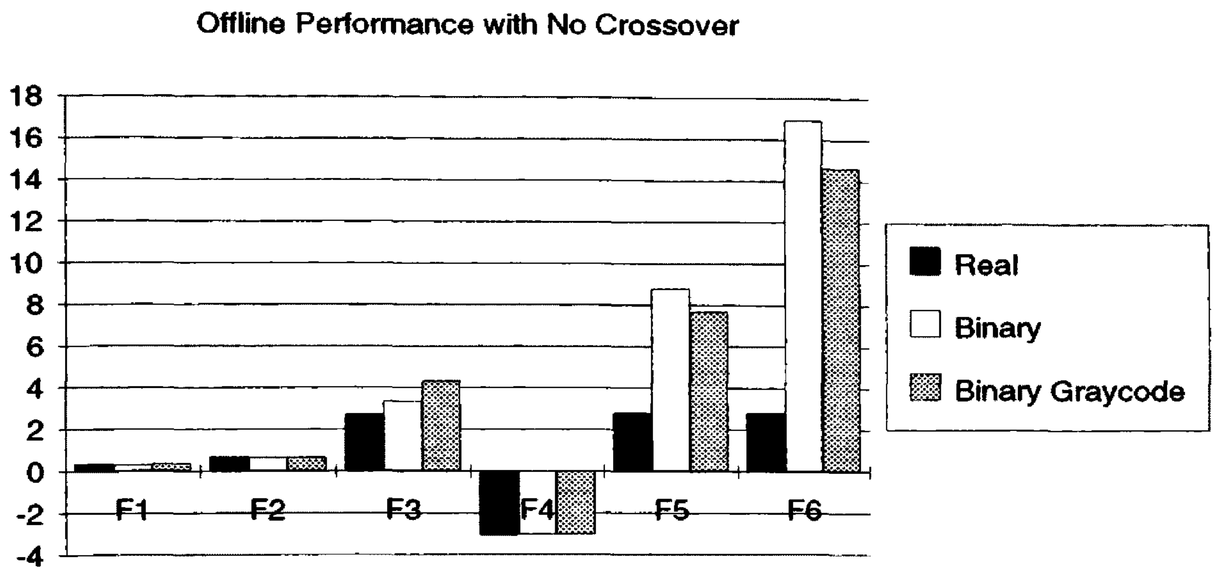

Figure 4.31

Examining the data for Offline performance with crossover in figure 4.32 reveals a very similar performance profile. In F1 through F4, the performance of the RealIndividual1 is very close to that of the no graycode experiment for BinaryIndividual. However, the performance recorded in F5 and F6 is again very different. This is evidence that for the first four deJong test problems, RealIndividual1 is successful in simulating a binary genetic algorithm.

The lack of success with functions F5 and F6 is interesting. RealIndividual1 out performs both binary and graycode versions of BinaryIndividual. This is possibly because of the continuous nature of the mutation defined within RealIndividual1. Further research in the future may explain the discrepancy.

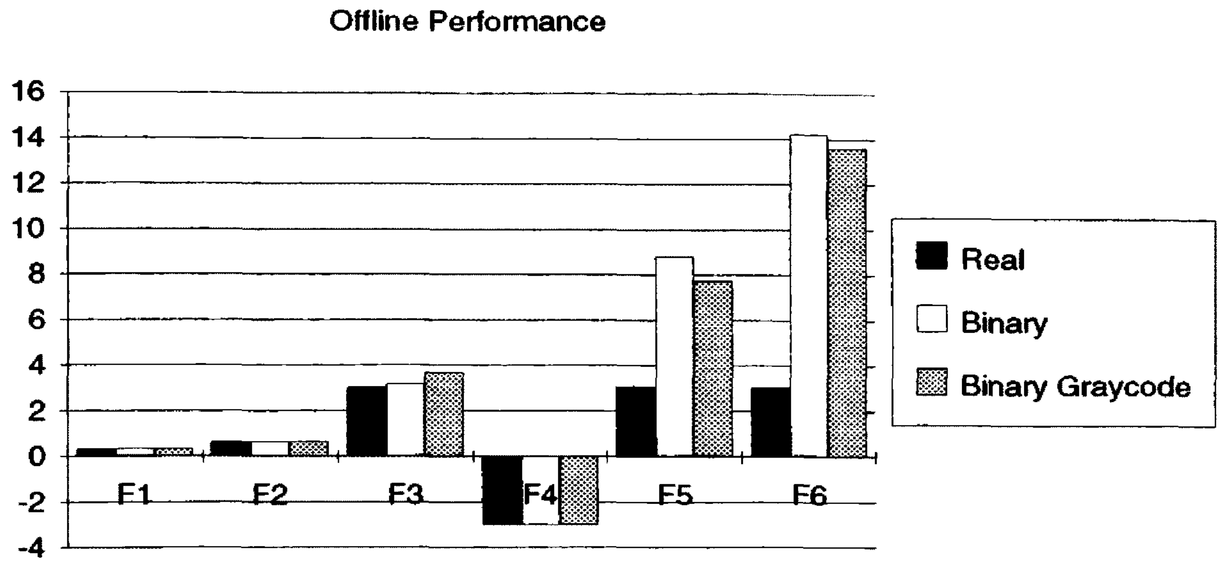

Figure 4.32

4.4. Adapting to Other Problem Types

The framework on which the OBJGEN program is based, is flexible enough to allow many different types of objective functions to be optimized. OBJGEN focuses on parametric objective functions with real parameters. This program does not need extensive modification to enable it to optimize other types of problems.

The first step in this process is to define the representation of the genetic material and then create an individual type that uses it. Creating a new type of individual will involve deriving a new class from the base class Individual. The representation of the genetic material will have to be coded into the derived class. The definitions of mutation, crossover and the other functions that Individual leaves undefined will also have to be supplied.

The second step is to create a function dispatcher for functions that accept the new genetic material representation as input and return a floating point number as a performance measure. This is done by deriving a new class from the class FunctionDispatcher. This is a simple operation because the function dispatcher does no processing of its own. It is just a repository for information about functions. It's only responsibility is dispensing the information on request.

The final step requires a simple modification to the OBJGEN mainline program. If there is only one variant of the new individual type, the prototype individual can be made directly in the mainline routine. This is done by defining a variable of the new individual type while passing pointers to the IteratorSet and the FunctionDispatcher to the constructor. In the case where there is more than one type of variant of the new individual class available, it is necessary to provide a facility that allows the user to select which variant to use. The RealIndividualManager can be used a guide for constructing a simple object that encapsulates the creation of a prototype individual.

As an example, consider the possibility of creating a type of individual where the genetic material is represented by a tree structure [6]. The objective function could accept the tree and evaluate it based on some criteria. The objective function would then return a value that would represent the performance of the individual.

```
class TreeIndividual : public Individual
{
  static TreeFunctionPointer Evaluator;

  Tree* Root;

  public:
  TreeIndividual (TreeFunctionDispatcher*, IteratorSet*);
  TreeIndividual ();
  ...
  virtual int  CalcPerformance (void);
  virtual Individual* Clone (void);
  virtual void Copy (Individual*);
  virtual void CrossOver (Individual*);
  virtual void Mutate (void);
  virtual void Print (FILE*);
  virtual int  operator== (Individual*);
  virtual void RandomSetup (void);
}
```

Figure 4.30

Here a TreeIndividual is defined as an individual that has a Tree as its genetic material. It uses an evaluation function that accepts a Tree as its input. The constructors need to fetch a pointer to the objective function from a function dispatcher. The pointer should be stored in the class variable Evaluator.

Each of the other functions must be defined as outlined in the section above about the class Individual. The RandomSetup routine would create a tree randomly. Mutation on a tree could be defined as adding or deleting nodes at random positions in the tree. Alternatively mutation could simply rearrange existing nodes in the tree. Crossover could exchange branches between the trees of two TreeIndividuals.

```
typedef double (*TreeFunctionPointer)(Tree*);
class TreeFunctionDispatcher : public FunctionDispatcher
{
  TreeFunctionPointer Functions;

  public:
  TreeFunctionDispatcher (int n) : FunctionDispatcher (n) {}
  void SetFunctions (Tree* f) { Functions = f; }
  TreeFunctionPointer GetFunction (int n) { return Functions[n]; }
};
```

Figure 4.31

This is the complete definition of a derivative of a FunctionDispatcher for objective functions requiring a pointer to a Tree as input. Initializing the list of functions is also very simple.

```
char TreeFunction1Name[] = "Function 1";
double TreeFunction1 (Tree* t)
{
  ...
}

char TreeFunction2Name[] = "Function 2";
double TreeFunction2 (Tree* t)
{
  ...
}

char* TreeFunctionNames[] = { TreeFunction1Name,
                              TreeFunction2Name };
TreeFunctionPointer TreeFunctions[] = { TreeFunction1,
                                        TreeFunction2 };

main (...)
{
  ...
  TreeFunctionDispatcher TFD(2);
  TFD.SetFunctionNames (TreeFunctionNames);
  TFD.SetFunctions (TreeFunctions);
  ...
  Individual* PrototypeIndividual = new TreeIndividual (&TFD, &IS);
  ...
}
```

Figure 4.32

In this example, two objective functions and their names are defined. Pointers to the functions are then collected into an array called TreeFunctions. Pointers to the names are also collect into an array called TreeFunctionNames. In the main line code, a TreeFunctionDispatcher called TFD is created. The constructor is passed the value of two, in order to prepare it to receive two functions. The following lines assign the function pointer array and the function name array to the function dispatcher. At this point, the function dispatcher is complete and ready to use. Its usage consists of merely passing it to the constructor of the prototype individual.

The other classes in OBJGEN will use the new individual type without modification. There is no need to alter the population classes, the probe classes or the Genetic class. These classes perform their functions on generic individuals without caring about the individuals' internal implementations.

5. Conclusion

The OBJGEN program and the framework on which it is based, are very robust. They can be applied to a wide variety of problems without needing major amounts of rewriting and recompiling. This robustness stems from the exploitation of the power of object oriented programming. The OOP features contribute to the construction and flexibility of Probes and Iterators.

The malleability of the program was of great assistance in devising the comparison of the two implementations. Since one program handled both types of individuals, there was no overhead involved in juggling two programs. Most of the code of the genetic algorithm served both types of individuals. All of the class hierarchies in OBJGEN work independently of the type of individual in use.

The ability of Probes to monitor the progress of the genetic algorithm without regard to the type of individual in current use, provided a uniform test bed environment. No special coding was necessary to provide the performance metrics of each type of individual. Probes provided a consistent report form.

The capabilities of Iterators to run most of the experiments in batches facilitated the experimental process. The experiments were set up and they ran with out any need for user intervention. The Iterators and Probes worked together to provide reports that were easy to analyze. Probes provided the data collection, while the iterators kept a record of the current parameter settings.

OBJGEN, when used with individuals of the class RealIndividual1 is successful in simulating a binary genetic algorithm with most of the deJong test problems. However, the success is muted by the performance discrepancies with the functions five and six. OBJGEN provides the tools for deeper research into the problem.

References

1Goldberg, David E. (1989) *Genetic Algorithms in Search Optimization and Machine Learning.* Addison-Wesley. p1

2Grefenstette, John J. (1984) GENESIS: A system for using geneticsearch procedures. *Procedings of the 1984 Conference on Intelligent Systems and Machines*

3 Baker, James E. (1987) Reducing bias and inefficiency in the selection algorithm. *Genetic Algorithms and Thier Applications: Procedings of the Second International Conference.* pp14-21

4 Wright, Alden H. (1990) Genetic Algorithms for Real Parameter Optimization

5 DeJong, K. A. (1975) "Analysis of the Behavior of a Class of Genetic Adaptive Systems," Ph.D. Dissertation, Department of Computer and Communications Sciences, University of Michiga, Ann Arbor, MI.

6 Koza, John R. (1990) "Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems," Computer Science Department, Stanford University, Stanford, CA.