Chapter 1
The GA Procedure (Experimental)

Chapter Contents

Chapter 1
The GA Procedure (Experimental)

Overview

Genetic algorithms are a family of local search algorithms that seek optimal solutions to problems using the principles of natural selection and evolution. Genetic algorithms can be applied to almost any optimization problem, and are especially useful for problems where other calculus-based techniques do not work, such as when the objective function has many local optima, is not differentiable or continuous, or solution elements are constrained to be integers or sequences. In most cases, genetic algorithms require more computation than specialized techniques that take advantage of specific problem structure or characteristics. However, for optimization problems with no such techniques available, genetic algorithms provide a robust general method of solution. This release of the GA procedure is experimental, and will be further developed and tested in later SAS releases. For the most recent updates to the documentation for this experimental procedure, see the Statistics and Operations Research Documentation page at http://support.sas.com/rnd/app/doc.html.

In general, genetic algorithms use some variation of the following procedure to search for an optimal solution:

initialization:	An initial population of solutions is randomly generated, and the objective function is evaluated for each member of this initial generation.
selection:	Individual members of the current generation are chosen stochastically either to parent the next generation or to be passed on to it, such that those members who are the fittest are more likely to be selected. A solution's fitness is based on its objective value, with better objective values reflecting higher fitness.
crossover:	Some of the selected solutions are passed to a crossover operator. The crossover operator combines two or more parents to produce new offspring solutions for the next generation. The crossover operator tends to produce new offspring that retain the common characteristics of the parent solutions, while combining the other traits in new ways. In this way, new areas of the search space are explored, hopefully while retaining optimal solution characteristics.
mutation:	Some of the next-generation solutions are passed to a mutation operator, which introduces random variations in the solutions. The purpose of the mutation operator is to ensure that the solution space is adequately searched to prevent premature convergence to a local optimum.

repeat:	The current generation of solutions is replaced by the new generation. If the stopping criterion is not satisfied, the process returns to the *selection* phase.

The crossover and mutation operators are commonly called *genetic operators*. Selection and crossover distinguish genetic algorithms from a purely random search and direct the algorithm toward finding an optimum. Mutation is designed to ensure diversity in the search to prevent premature convergence to a local optimum.

There are many ways to implement the general strategy just outlined, and it is also possible to combine the genetic algorithm approach with other heuristic solution improvement techniques. In the traditional genetic algorithm, the solutions space is comprised of bit-strings, mapped to an objective function, and the genetic operators are modeled after biological processes. Although there is a theoretical foundation for the convergence of genetic algorithms formulated in this way, in practice most problems do not fit naturally into this paradigm. Modern research has shown that optimizations can be set up using the natural solution domain (for example, a real vector or integer sequence) and applying crossover and mutation operators analogous to the traditional genetic operators, but more appropriate to the natural formulation of the problem. This is the approach, sometimes called *evolutionary computing*, taken in the GA procedure. It enables you to model your problem using a variety of solution forms including sequences, integer or real vectors, boolean encodings, and combinations of these. The GA procedure also provides you with a choice of genetic operators appropriate for these encodings, while allowing you to write your own.

The GA procedure enables you to implement the basic genetic algorithm by default, and also to employ other advanced techniques to handle constraints, accelerate convergence, and perform multiobjective optimizations. These advanced techniques are discussed in the "Details" section beginning on page 43.

Although genetic algorithms have been demonstrated to work well for a variety of problems, there is no guarantee of convergence to a global optimum. Also, the convergence of genetic algorithms can be sensitive to the choice of genetic operators, mutation probability, and selection criteria, so that some initial experimentation and fine-tuning of these parameters is often required.

Getting Started

The optimization problem is described using programming statements, which initialize problem data and specify the objective, genetic operators, and other optimization parameters. The programming statements are executed once, and are followed by a RUN statement to begin the optimization process. The GA procedure enables you to define subroutines and designate them to be called during the optimization process to calculate objective functions, perform genetic mutation or crossover operations, or monitor and control the optimization. All variables created within a subroutine are local to that routine; to access a global variable defined within the GA procedure, the subroutine must have a parameter with the same name as the variable.

To set up a genetic algorithm optimization, your program needs to perform the following steps:

1. You must first initialize your problem data, such as cost coefficients and parameter limits.
2. You must then specify five basic optimization parameters:

• *Encoding*:	The general structure and form of the solution
• *Objective*:	The function to be optimized
• *Selection*:	How members of the current solution generation are chosen to propagate the next generation
• *Crossover*:	How the attributes of parent solutions are combined to produce new offspring solutions
• *Mutation*:	How random variation is introduced into the new offspring solutions to maintain genetic diversity

3. Next, you need to generate a population of solutions for the initial generation.
4. Finally, you need to control the execution of the algorithm and record your results.

The following sections discuss each of these items in detail.

Initializing the Problem Data

The GA procedure offers great flexibility in how you initialize the problem data. You can either read data from SAS data sets that are created from other SAS procedures and DATA steps, or you can initialize the data with programming statements.

In the PROC GA statement, you can specify up to five data sets to be read with the DATAn= option, where n is a number from 1 to 5, that can be used to initialize parameters and data vectors applicable to the optimization problem. For example, weights and rewards for a Knapsack Problem could be stored in the variables WEIGHT and REWARD in a SAS data set. If you specify the data set with a DATA1= option, the arrays WEIGHT and REWARD are initialized at the start of the procedure and are available for computing the objective function and evaluating the constraints with program statements. You could store the number of items and weight limit constraint in another data set, as illustrated in the sample programming statements that follow.

```
data input1;
   input weight reward;
   datalines;
1    5
2    3
4    7
1    2
8    3
```

```
6    9
2    6
4    3
...
;

data input2;
   input nitems limit;
   datalines;
10  20
;

proc ga data1 = input1  /* creates arrays weight and reward */
        data2 = input2; /* creates variables nitems and limit */

function objective( selected[*], reward[*], nitems);
   array x[1] /nosym;
   call dynamic_array(x, nitems);
   call ReadMember(selected,x,1);
   obj = 0;
   do i=1 to nitems;
     obj = obj + reward[x[i]];
   end;
   return(obj);
endsub;

[Other statements follow]
```

With these statements, the DATA1= option first establishes the arrays weight and reward from the data set input1, and the DATA2= option causes the variables nitems and limit to be created and initialized from the data set input2. The reward array and the nitems variable are then used in the objective function.

For convenience in initializing two-dimensional data such as matrices, the GA procedure provides you with the MATRIXn= option, where n is a number from 1 to 5. A two-dimensional array is created within the GA procedure with the same name as the option, containing the numeric data in the specified data set. For example, a table of distances between cities for a Traveling Salesman Problem could be stored as a SAS data set, and a MATRIX1= option specifying that data set would cause a two-dimensional array named MATRIX1 to be created containing the data at the start of the GA procedure.

```
data distance;
   input d1-d10;
   datalines;
0 5 3 1 2 ...
5 0 4 2 6 ...
3 4 0 1 3 ...
...
;
```

```
proc ga matrix1 = distance;
ncities = 10;
call SetEncoding('S10');
call SetObjTSP(matrix1);

[Other statements follow]
```

In this example, the data set distance is used to create a two-dimensional array matrix1, where matrix1$[i, j]$ is the distance from city i to city j. The SetObjTSP call then passes matrix1 to the Traveling Salesman Problem objective function. Note that when a MATRIXn= option is used, the names of variables in the data set are not transferred to the GA procedure as they are with a DATAn= option; only the numeric data is transferred.

You can also initialize problem data with programming statements. The programming statements in the GA procedure are executed before the optimization process begins. The variables created and initialized can be used and modified as the optimization progresses. The programming statement syntax is much like the SAS DATA step, with a few differences (see the "Syntax" section beginning on page 20). Special calls are described in the next sections that enable you to specify the objective function and genetic operators, and to monitor and control the optimization process. In the following code, a two-dimensional matrix is set up with programming statements to provide the distances for a 10-city symmetric Traveling Salesman Problem, between locations specified in a SAS data set.

```
data positions;
   input x y;
   datalines;
100 230
50  20
150 100
...
;

proc ga data1 = positions;

call SetEncoding('S10');
ncities = 10;

array distances[10,10] /nosym;

do i = 1 to ncities;
   do j = 1 to i;
      distances[i,j] = sqrt((x[i]-x[j])**2 + (y[i] - y[j])**2);
      distances[j,i] = distances[i,j];
   end;
end;

call SetObjTSP(distances);
```

In this example, the DATA1= option creates arrays x and y containing the coordinates of the cities in an x-y grid, read in from the positions data set. An array programming statement creates a matrix of distances between cities, and the loops calculate Euclidean distances from the position data. The array statement is used to create internal data vectors and matrices. It is similar to the array statement used in the SAS DATA step, but the /NOSYM option is used in this example to set up the array without links to other variables. This option allows the array elements to be indexed more efficiently and the array to be passed efficiently to subroutines. You should use the /NOSYM option whenever you are creating an array that might be passed as an argument to a function or call routine.

Choosing the Problem Encoding

Problem encoding refers to the structure or type of solution space that is to be optimized, such as real-valued fixed-length vectors or integer sequences. The GA procedure offers encoding options appropriate to several types of optimization problems. You specify the problem encoding with a SetEncoding call statement,

call SetEncoding(‘*encoding*’**);**

where the *encoding* string is a letter followed by a number, which specifies the type of encoding and the number of elements. The allowed letters and corresponding types of encoding are

R or *r*:	Real-valued vector. This type of encoding is used for general nonlinear optimization problems.
I or *i*:	Integer-valued vector. This encoding is used for integer-valued problems. An example might be an assignment problem, where the positions within the vector represent different tasks, and the integer values represent different machines or other resources that might be applied to each task.
B or *b*:	Boolean vector. Each element represents one bit or true/false value.
S or *s*:	Sequence or permutation. In this encoding, each solution is composed of a sequence of integers ranging from 1 to the number of elements, with different solutions distinguished by different orderings of the elements. This encoding is commonly used for routing problems such as the Traveling Salesman Problem or for scheduling problems.

For example, the following statement specifies a 10-element integer vector encoding:

```
call SetEncoding('I10');
```

For problems where the solution form requires more than one type of encoding, you can specify multiple encodings in the *encoding* string. For example, if you want to optimize the scheduling of 10 tasks and the assignment of resources to each task, you could use a *segmented* encoding:

```
call SetEncoding('I10S10');
```

Here the I10 (10-element integer vector) is assigned to the first segment, and represents the resource assignment. The S10 (10-element sequence) is assigned to a second segment, and represents the sequence of tasks. The use of segmented encodings is described in the "Using Multisegment Encoding" section on page 43.

Setting the Objective Function

Before executing a genetic algorithm, you must specify the objective function to be optimized. This is done with a SetObjFunc call:

call SetObjFunc('*fname*', *minmax***);**

where *fname* is the name of an objective function you define in your input (see the "Defining an Objective Function" section on page 55), and *minmax* is a number set to 0 to specify minimization or 1 to specify maximization. The GA procedure also provides a predefined objective function you can use for the Traveling Salesman Problem:

call SetObjTSP(*distances***);**

where *distances* is a two-dimensional array giving the distances between nodes. Other common objective functions will be added in future releases.

Controlling the Selection Process

There are two competing factors that need to be balanced in the selection process: the *selective pressure* and *genetic diversity*. Selective pressure, the tendency to select only the best members of the current generation to propagate to the next, is required to direct the genetic algorithm to an optimum. Genetic diversity, the maintenance of a diverse solution population, is also required to ensure that the solution space is adequately searched, especially in the earlier stages of the optimization process. Too much selective pressure can lower the genetic diversity so that the global optimum is overlooked and the genetic algorithm converges prematurely. Yet, with too little selective pressure, the genetic algorithm may not converge to an optimum in a reasonable time. A proper balance between the selective pressure and genetic diversity must be maintained for the genetic algorithm to converge in a reasonable time to a global optimum.

The GA procedure offers two versions of a standard technique for the selection process commonly known as *tournament selection*. In general, the tournament selection process randomly chooses a group of members from the current population, compares their fitness, and selects the fittest from the group to propagate to the next generation. Tournament selection is one of the fastest selection methods, and offers good control over the selection pressure.

In the first version of tournament selection, you can control the selective pressure by specifying the tournament size, the number of members chosen to compete in each tournament. This number should be 2 or greater, with 2 implying the weakest selection pressure. Tournament sizes from 2 to 10 have been successfully applied to various genetic algorithm optimizations, with sizes over 4 or 5 considered to represent strong selective pressure. This selection option is chosen with the following SetSel call:

```
call SetSelTournament(size);
```

where *size* is the desired tournament size.

The second version of tournament selection provides weaker selective pressure than the first version just described. The tournament size is set at 2, and the fittest participant is selected with a probability that you specify. This fittest-is-selected probability can range from 0.5 to 1.0, with 1.0 implying that the best member is always chosen (equivalent to a conventional tournament of size 2) and 0.5 implying an equal chance of either member being chosen (equivalent to pure random selection). This selection option is chosen with the following SetSel call:

```
call SetSelDuel(p);
```

where p is the fittest-is-selected probability.

One potential problem with tournament selection is that it does not guarantee that the best solution in the current generation is passed on to the next. To resolve this problem, the GA procedure enables you to specify an *elite* parameter, which ensures that the very best solutions are passed on to the next generation unchanged by mutation or crossover. Use the SetElite call:

```
call SetElite(elite);
```

where *elite* is an integer greater than or equal to 0. The GA procedure preserves the *elite* best solutions in the current generation, and ensures they are passed on to the next generation unchanged. When writing out the final solution population, the first *elite* members are the best of the generation, and are sorted by their fitness, such that the fittest is first. By default, if you do not call SetElite in your program, an *elite* value of 1 is used. Setting the *elite* parameter to a higher number accelerates the convergence of the genetic algorithm; however, it may also lead to premature convergence before reaching a global optimum, so it should be used with care.

In the future, other selection methods such as roulette and rank selection may be offered as options. If you do not call SetSel in your input, then the default behavior for the GA procedure is to use the first version of tournament selection, with a tournament size of 2.

Setting Crossover Parameters

There are two crossover parameters that need to be specified: the crossover probability and the crossover operator. Members of the current generation that have passed the selection process either go to the crossover operator or are passed unchanged into the next generation, according to the crossover probability. To set the probability, you use a SetCrossProb call statement:

```
call SetCrossProb(prob);
```

where *prob* is a real number between 0 and 1. A value of 1 implies that the crossover operator is always applied, while 0 effectively turns off crossover. If you don't explicitly set the crossover probability with this call, a default value of 1 is used.

To set the crossover operator, you use a SetCross call. To supply your own operator, use

call SetCrossRoutine('*name*');

where *name* is the name of your crossover subroutine. The GA procedure also makes available to you several standard crossover operators appropriate for each type of encoding. See the "Crossover Operators" section beginning on page 44 for more detail on each operator. These operators can be specified as follows.

For Boolean, Real, and Integer encodings:

call SetCrossSimple(*alpha*);

call SetCross2Point(*alpha*);

call SetCrossUniform(*alpha*);

For Integer and Real encodings:

```
call SetCrossArithmetic();
```

For Real encoding only:

```
call SetCrossHeuristic();
```

For Sequence encoding:

```
call SetCrossOrder();

call SetCrossPMatch();

call SetCrossCycle();
```

If you do not use a SetCross call to set the crossover operator, the GA procedure uses a default crossover operator, which depends on the encoding.

Setting Mutation Parameters

There are two mutation parameters: the mutation probability and the mutation operator. Members of the next generation are chosen to undergo mutation with the mutation probability you specify. To set the probability, you use a SetMutProb call statement:

call SetMutProb(*prob*);

where *prob* is a real number between 0 and 1. This probability is usually fairly low (0.05 or less), since mutation tends to slow the convergence of the genetic algorithm. If you don't explicitly set the mutation probability with this call, a default value of 0.05 is used.

To set the mutation operator, you use a SetMut call. To supply your own operator, use

call SetMutRoutine(‘*name*’**);**

where *name* is the name of your mutation subroutine. The GA procedure also makes available to you several standard mutation operators appropriate for each type of encoding. See the “Mutation Operators” section beginning on page 50 for more detail on each operator. These operators can be specified as follows.

For Real and Integer encodings:

call SetMutDelta(*delta*, *n***);**

For Boolean, Real, and Integer encodings:

call SetMutUniform(*n***);**

For Sequence encoding:

call SetMutSwap(*n***);**

call SetMutInvert();

If you do not use a SetMut call to set the mutation operator, the GA procedure uses a default mutation operator, which depends on the encoding.

Creating the Initial Generation

The last step in the initialization for the genetic algorithm optimization is to create the initial solution population, the first generation. The GA procedure provides two methods for generating the initial population. You can specify a data set in the FIRSTGEN= option of the PROC GA statement that is read to populate the initial generation, or you can use an initialize call:

call Initialize(‘*initializer*’, *PopulationSize***);**

where *initializer* is either the name of a subroutine you have defined to create new solutions, or DEFAULT to have the GA procedure perform a default initialization.

The default initialization action performed for each solution segment depends on its encoding. For sequence encoding, the default action produces randomly shuffled integer sequences from 1 to the segment size. For boolean encoding, the default action is to randomly generate 0 or 1 bits for each element. For real and integer encodings, the default initialization depends on whether bounds have been specified for the segment with a SetBounds call. If bounds have been set for the segment, then values randomly distributed between the bounds are generated. If no bounds have been set, then the segment is filled with 0 values. If all the segments in the encoding are real or integer, default initialization is only allowed if at least one segment has bounds specified by a SetBounds call.

For other cases where the solution space has more complicated bounds, you are required to supply your own subroutine to generate new solutions. The specifications for this subroutine are discussed in the “Defining a User Initialization Routine” section on page 56.

Monitoring Progress and Reporting Results

The GA procedure enables your program to monitor and alter parameters during the optimization process and record final results.

If a data set is specified in the LASTGEN= option of the PROC GA statement, then the last generation of solutions is written out to the data set. See the "Syntax" section beginning on page 20 for a description of the data set created by the LASTGEN= option.

You can define a subroutine and designate it to be called at each iteration in an update phase, which occurs after the evaluation phase and before selection, crossover, and mutation. Your subroutine can check solution values and update and store variables you have defined, adjust any of the optimization parameters such as the mutation probability or *elite* value, or check termination criteria and end the optimization process. An update routine can be especially helpful in implementing advanced techniques such as multiobjective optimization. You can specify an update subroutine with a SetUpdateRoutine call:

call SetUpdateRoutine('*routine*'**);**

where *routine* is the name of your subroutine to be executed at each iteration.

You can set the maximum number of iterations allowed for the optimization process with the MAXITER= option in the PROC GA statement. If none is specified, a default value of 500 iterations is used. You can also control the number of iterations dynamically in your program, using the ContinueFor call:

call ContinueFor(n**);**

where n is the number of iterations to allow beyond the current iteration. A value of 0 ends the optimization process at the current iteration. One common way this call might be used is to include it in the logic of an update subroutine declared in the SetUpdateRoutine call. The update subroutine could check the objective values, and end the optimization process when the optimal value of the objective function has not improved for a specific number of iterations. A ContinueFor call overrides an iteration limit set with the MAXITER= option.

To perform post-processing of the optimization data, you can use a SetFinalize call to instruct the GA procedure to call a subroutine you have defined, after the last iteration:

call SetFinalize('*routine*'**);**

where *routine* is the name of a subroutine you have defined. Your finalize subroutine could perform some post-processing tasks, such as applying heuristics or a local optimization technique to try to improve the final solution.

A Simple Example

This example illustrates the application of genetic algorithms to function optimization over a real-valued domain. It finds the minimum of the Shubert function:

$$\left[\sum_{i=1}^{5} i\cos\left[(i+1)x_1+i\right]\right]\left[\sum_{i=1}^{5} i\cos\left[(i+1)x_2+i\right]\right]$$

where $-10 \le x_i \le 10$ for $i = 1, 2$.

```
proc ga seed = 12 maxiter = 30;

/* the objective function to be optimized */
function shubert(selected[*]);
   array x[2] /nosym;
   call ReadMember(selected,1,x);
   x1 = x[1];
   x2 = x[2];
   sum1 = 0;
   do i = 1 to 5;
      sum1 = sum1 + i * cos((i+1)* x1 + i);
   end;
   sum2 = 0;
   do i = 1 to 5;
      sum2 = sum2 + i * cos((i+1) * x2 + i);
   end;
   result = sum1 * sum2;
   return(result);
endsub;

/* Set the problem encoding */
call SetEncoding('R2');

/* Set upper and lower bounds on the solution components */
array LowerBound[2] /nosym (-10 -10);
array UpperBound[2] /nosym (10 10);
call SetBounds(LowerBound, UpperBound);

/* Set the objective function */
call SetObjFunc('shubert',0);

/* Set the crossover parameters */
call SetCrossProb(0.65);
call SetCrossHeuristic();

/* Set the mutation parameters */
call SetMutProb(0.15);
array delta[2] /nosym (0.2 0.2);
call SetMutDelta(delta, 1);

/* Set the selection criteria */
call SetSelTournament(2);
call SetElite(2);

/* Initialize the first generation, with 120 random solutions  */
call Initialize('DEFAULT',120);
```

```
/* Now execute the Genetic Algorithm */
run;
quit;
```

At the beginning of the program, the PROC GA statement sets the initial random number seed and sets the maximum number of iterations to 30.

A routine to compute the objective function (`function shubert`) is then defined. This function is called by the GA procedure once for each member of the solution population at each iteration. Note that the GA procedure passes the array selected as the first parameter of the function, and the function uses that array to obtain the selected solution elements with a ReadMember call, which places the solution in the array x. The second parameter of the ReadMember call is 1, specifying that segment 1 of the solution be returned, which in this case is the only segment. The programming statements that follow compute the value of the objective function and return it to the GA procedure.

After the function definition, the 'R2' passed to the SetEncoding call specifies that solutions are single-segment, with that segment containing two elements that are real-valued. Next, a lower bound of -10 and upper bound of 10 are set for each solution element with the SetBounds call. The SetObjFunc call specifies the previously defined Shubert function as the objective function for the optimization; the second parameter value of 0 indicates that a minimum is desired. The SetCrossProb call sets the crossover probability to 0.65, which means that, on average, 65% of the solutions passing the selection phase will undergo the crossover operation. The crossover operator is set to the heuristic operator by the SetCrossHeuristic call. Similarly, the mutation probability is set to 0.15 with the SetMutProb call, and the delta operator is set as the mutation operator with the SetMutDelta call. The selection criteria are then set: a conventional tournament of size 2 is specified with SetSelTournament call, and an *elite* value of 2 is specified with the SetElite call. The *elite* value implies that the best two solutions of each generation are always carried over to the next generation unchanged by mutation or crossover. The last step before beginning the optimization is the Initialize call. This call sets the population size at 120, and specifies the default initialization strategy for the first population. For real encoding, this means that an initial population randomly distributed between the upper and lower bounds specified in the SetBounds call is generated. Finally, when the RUN statement is encountered, the GA procedure begins the optimization process. It iterates through 30 generations, as set by the MAXITER= option.

The Shubert function has 760 local minima, 18 of which are global minima, with a minimum of -186.73. If you experiment with different random seeds with the SEED= option, PROC GA generally converges to a different global minimum each time. Figure 1.1 shows the output for the chosen seed.

```
                    PROC GA Optimum Values

                           Objective

                         -186.7307143


                           Solution
                   Element            Value

                         1     4.8579319191
                         2     5.4831317424
```

Figure 1.1. Shubert Function Example Output

Syntax

To initialize your data and describe your model, you use program statements with a syntax similar to the SAS DATA step, augmented with some special function calls to communicate with the genetic algorithm optimizer. Most of the program statements used in the SAS DATA step can be used in the GA procedure, and these are described fully in the *SAS Language Guide* and base SAS documentation. Below is an alphabetical list of the statements and special function calls used.

PROC GA *options* ;
- **ContinueFor Call**;
- **Cross Calls**;
- **Dynamic_array Call**;
- **EvaluateLC Call**;
- **GetDimensions Call**;
- **GetObjValues Call**;
- **GetSolutions Call**;
- **Initialize Call**;
- **MarkPareto Call**;
- **Mut Calls**;
- **PackBits Call**;
- **Program Statements**;
- **ReadChild Call**;
- **ReadCompare Call**;
- **ReadMember Call**;
- **ReadParent Call**;
- **ReEvaluate Call**;
- **SetBounds Call**;
- **SetCross Calls**;
- **SetCrossProb Call**;
- **SetElite Call**;
- **SetEncoding Call**;
- **SetFinalize Call**;

SetMut Calls;
SetMutProb Call;
SetObj Calls;
SetSel Calls;
SetUpdateRoutine Call;
UnpackBits Function;
UpdateSolutions Call;
WriteChild Call;
WriteMember Call;

PROC GA Statement

invokes the GA procedure

PROC GA *options* **;**

The following options are used with the PROC GA statement.

DATAn**=***SAS-data-set*
specifies a data set containing data required to specify the problem, where n is an integer from 1 to 5. The data set is read and variables created matching the variables of the data set. If the data set has more than one observation, then the newly created variables are vector arrays with the size equal to the number of observations.

FIRSTGEN=*SAS-data-set*
specifies a SAS data set containing the initial solution generation. Different segments in the solution should be identified by variable names consisting of a letter followed by numbers representing the elements in the segments, in alphabetical order. For example, if the first segment of the solution uses real encoding and contains 10 elements, it should be represented by numeric variables A1, A2, . . . , A10. A second segment with integer encoding and five elements would be specified in variables B1, B2, . . . , B5. To save memory for segments with boolean encoding, up to 32 binary elements are packed into each variable, with the least significant bit in the variable corresponding to the first element in the segment. For example, if the third segment of the solution was boolean encoding with 40 elements, it should be contained in variables C1 and C2, with C1 containing the first 32 elements, and C2 containing elements 33 through 40. The FIRSTGEN= and LASTGEN= options are designed to work together, so that a data set generated with a LASTGEN= option can be specified in the FIRSTGEN= option of a later run of the GA procedure.

MATRIXn**=***SAS-data-set*
specifies a data set containing two-dimensional matrix data, where n is an integer from 1 to 5. A two-dimensional numeric array with the same name as the option is created and initialized from the data set. This option is provided to facilitate the input of tabular data to be used in setting up the optimization problem. Examples of data that might be provided by this option include a distance matrix for a Traveling Salesman Problem or a matrix of coefficients for linear constraints.

MAXITER=n
specifies the maximum number of iterations to allow for the optimization process. A ContinueFor call overrides a limit set by this option.

LASTGEN=*SAS-data-set*

specifies a SAS data set into which the final solution generation is written. Different segments in the solution are identified by variable names consisting of a letter followed by numbers representing the elements in the segments, in alphabetical order. For example, if the first segment of the solution uses real encoding and contains 10 elements, it would be represented by numeric variables A1, A2, . . . , A10. A second segment with integer encoding and five elements would be specified in variables B1, B2, . . . , B5. To save memory for segments with boolean encoding, up to 32 binary elements are packed into each variable, with the least significant bit in the variable corresponding to the first element in the segment. For example, if the third segment of a solution was boolean encoding with 40 elements, it would be contained in variables C1 and C2, with C1 containing the first 32 elements, and C2 containing elements 33 through 40. In addition to the solutions elements, the final objective value for each solution is output in the OBJECTIVE variable. The FIRSTGEN= and LASTGEN= options are designed to work together, so that a data set generated with a LASTGEN= option can be specified in the FIRSTGEN= option of a later run of the GA procedure.

SEED=n

specifies an initial seed to begin random number generation. This option is provided for reproducibility of results. If it is not specified, or if it is set to 0, a seed is chosen based on the system clock.

ContinueFor Call

sets the number of additional iterations for the genetic algorithm optimization

call ContinueFor(*niter* **);**

The input to the ContinueFor subroutine is as follows:

niter	specifies that the optimization continues for *niter* more iterations. To stop at the current iteration, set *niter* to 0.

Cross Calls

executes a genetic crossover operator from within a user subroutine

This call can take one of several forms:

call CrossSimple(*selected, seg, alpha* **);**

call Cross2Point(*selected, seg, alpha* **);**

call CrossUniform(*selected, seg, alpha* **);**

call CrossArithmetic(*selected, seg* **);**

call CrossHeuristic(*selected, seg* **);**

call CrossOrder(*selected, seg* **);**

call CrossPMatch(*selected, seg* **);**

call CrossCycle(*selected, seg* **);**

The inputs to the subroutine are as follows:

selected	is an array that specifies the solutions to undergo crossover.
seg	is the desired segment of the solutions to which the crossover operation should be applied.
alpha	is used only with the Simple, 2point and Uniform operators, and must be a value between 0 and 1.

The Cross subroutines should only be called from within a user crossover subroutine. A user subroutine is only required if multisegment encoding is used, or if you want to modify the action of a standard crossover operator. For simple single-segment encoding, you would normally use a SetCross call to set the crossover operator without the need to define your own subroutine. The precise action of these crossover operators is described in the "Crossover Operators" section beginning on page 44. The *alpha* parameter is only used for real and integer encodings; it controls the amount of change between parent and child solutions. A low value of *alpha* restrains the operator so that the offspring look very much like the parents, while a value of 1 allows complete recombination. The value *alpha* = 1 corresponds to the classic crossover operators used in early genetic algorithm development. For boolean encoding, *alpha* is ignored, and is effectively 1.

Dynamic_array Call

allocates a numeric array

call Dynamic_array(*arrayname, dim1<, dim2, ..., dim6>* **);**

The inputs to the Dynamic_array call are as follows:

arrayname	is a previously declared array, whose dimensions are to be reallocated.
dim1	is the size of the first dimension.
dim2,...,dim6	are optional additional dimensions. Up to six dimensions may be specified.

The Dynamic_array call is normally used to allocate working arrays when the required size of the array is data-dependent. It is often useful in user routines for genetic operators or objective functions to avoid hard-coding array dimensions that might depend on segment length or population size. The array to be allocated must first be declared in an ARRAY statement with the expected number of dimensions, as in the following example.

```
subroutine sub(nx, ny);
   array x[1] /nosym;
   call dynamic_array(x, nx);
   array xy[1,1] /nosym;
   call dynamic_array(xy, nx, ny);
   ...
```

EvaluateLC Call

evaluates linear constraints

call EvaluateLC(*lc, results, sum, selected, seg<, child>* **);**

The inputs to the EvaluateLC subroutine are as follows:

lc	is a two-dimensional array representing the linear constraints.
results	is a numeric array to receive the magnitude of the constraint violation for each linear constraint.
sum	is a variable to receive the sum of the constraint violations over all the constraints.
selected	is an array identifying the selected solution.
seg	is the segment of the solution to which the linear constraints apply.
child	is an optional parameter, and should only be specified when EvaluateLC is called from a user crossover operator.

The EvaluateLC routine can be called from a user crossover operator, mutation operator, or objective function to determine if a solution violates linear inequality constraints of the form $Ax \le b$. For n linear constraints in m variables, the *lc* array should have dimension $n \times (m+1)$. For each linear constraint $i = 1, \ldots, n$, *lc*$[i, j] = A[i, j]$ for $j = 1, \ldots, m$, and *lc*$[i, m+1] = b[i]$. The *results* array should be one-dimensional with size n. The EvaluateLC call fills in the elements of *results* such that

$$\textit{results}[i] = \begin{cases} 0, & \text{if } \sum\limits_{j=1}^{m} A[i,j]x[j] \le b[i] \\ \sum\limits_{j=1}^{m} A[i,j]x[j] - b[i], & \text{otherwise} \end{cases}$$

In the variable *sum*, the EvaluateLC call returns the value $\sum_{i=1}^{n}$ *results*$[i]$. Note that *sum* ≥ 0, and *sum*$= 0$ implies no constraints are violated. When you call EvaluateLC from your user routine, the *selected* parameter of the EvaluateLC call must be the same as the first parameter passed to your user routine to properly identify the solution to be checked. The *seg* parameter identifies which segment of the solution should be checked. Real, integer, or boolean encodings can be checked with this routine. If EvaluateLC is called from a user crossover operator, the *child* parameter must be specified to indicate which offspring is to be checked. The value *child* = 1 requests the first offspring, *child* = 2 requests the second, and so on.

GetDimensions Call

gets the dimensions of an array variable

call GetDimensions(*source, dest* **);**

The inputs to the GetDimensions subroutine are as follows:

source	is the array variable whose dimensions are desired.
dest	is an array to receive the dimensions of *source*.

The GetDimensions subroutine is used to get the dimensions of an array passed into a user subroutine. The input *dest* should have at least as many dimensions as *source*. Any extra dimensions in *dest* are filled with zeros.

GetObjValues Call

retrieves objective function values from the current solution generation

call GetObjValues(*dest, n* **);**

The inputs to the GetObjValues subroutine are as follows:

dest	is an array to receive the objective values.
n	is the number of objective values to get.

The GetObjValues subroutine is used to retrieve the objective values for the current solution generation. If the *elite* parameter from a SetElite call is 1 or greater, then the first *elite* members of the population are the fittest of the population, and they are sorted in order, starting with the most fit. The input *dest* should be a dimensioned variable, with dimension greater than or equal to *n*.

GetSolutions Call

retrieves solutions from the current generation

call GetSolutions(*sol, n, seg* **);**

The inputs to the GetSolutions subroutine are as follows:

sol	is an array to receive the solution elements.
n	is the number of solutions to get.
seg	is the segment of the solution to retrieve.

The GetSolutions subroutine is used to retrieve solutions from the current generation. You would normally call it from an update or finalize subroutine for post-processing or analysis. If the *elite* parameter has been set with a SetElite call, then the first *elite* members of the population are the fittest, and they are sorted in order, starting with the most fit. The *sol* variable should have two dimensions, with the first dimension representing the solution number, and the second representing the element within the solution. For example, if the encoding of the problem was I10, then $sol[2, 3]$ would be the value of the third element of the second solution in the current population. For real, integer, and sequence encoding, each solution element is mapped to the corresponding element of the *sol* array. For boolean encoding, the bits are packed into the array, and you should use the PackBits and UnpackBits functions to manipulate individual bits. The *seg* parameter specifies the solution segment desired. For example, if the encoding was set in the SetEncoding call to 'R10I5', then segment 1 is R10 and segment 2 is I5.

Initialize Call

creates the initial solution generation

call Initialize(*'initializer', size* **);**

The inputs to the Initialize subroutine are as follows:

initializer	is a string that specifies the initialization strategy.
size	is the size of the initial solution generation.

The Initialize subroutine must be called to create the first solution generation. If the *initializer* parameter is DEFAULT, then the GA procedure initializes solution segments with a default action appropriate to the problem encoding. If the FIRSTGEN= option is specified in the PROC GA statement, then the default is to read in *size* solutions directly from the data set. If there is no FIRSTGEN= option specified, then for sequence encoded segments, the default action is to generate random sequences of numbers from 1 to the segment size, and for boolean encoded segments, random bit vectors are generated. For integer and real encoded segments, the default action is to generate vectors randomly distributed between the bounds specified in a SetBounds call, or if SetBounds has not been called, the segment is filled with zeros. If all the solution segments are real or integer, then at least one must have bounds set in order to specify the default action.

If *initializer* is not DEFAULT, then it must be the name of a subroutine you have defined through program statements. The first parameter of your subroutine must be a numeric array. The GA procedure calls your subroutine *size* times, passing in an array in the first parameter that specifies the selected solution to be initialized. You should use this parameter in a WriteMember call to assign initial values to the solution segments desired. See the "Defining a User Initialization Routine" section on page 56 for more information on defining an initialization subroutine.

MarkPareto Call

identifies the Pareto-optimal set from a population of solutions

call MarkPareto(*result, n, objectives, minmax* **);**

The inputs to the MarkPareto call are as follows:

result	is a one-dimensional array to accept the results of the evaluation. Its size should be the same as the size of the population being evaluated; *result*$[i] = 1$ if solution i is Pareto-optimal, and 0 otherwise.
n	is a variable to receive the number of Pareto-optimal solutions.
objectives	is a two-dimensional array that contains the multiple objective values for each solution. It should be dimensioned $[p, q]$, where p is the size of the population, and q is greater than or equal to the number of objectives to be considered.
minmax	is a one-dimensional array to specify how the objective values are to be used. It should be of size q, where q is greater than or equal to the number of objectives to be considered. *minmax*$[k] = -1$ if objective k is to be minimized, *minmax*$[k] = 1$ if objective k is to be maximized, *minmax*$[k] = 0$ if objective k is not to be considered, and *minmax*$[k] = -2$ designates an objective that prevents the member from being considered for Pareto-optimality if it is nonzero.

The MarkPareto call is used to identify the Pareto-optimal subset from a population of solutions. See the "Optimizing Multiple Objectives" section on page 59 for a full discussion of Pareto-optimality. MarkPareto can be called from a user update routine, which is called after the individual solution objective values have been calculated, and before selection takes place. To make best use of this routine, in your encoding you need to set up a segment to record all the objective values you intend to use in the Pareto-optimal set evaluation. In a user objective function, you should calculate the multiple objectives and write them to the chosen segment. In an update routine (designated with a SetUpdateRoutine call), you can use a GetSolutions call to retrieve these segments, and then pass them to a MarkPareto call. The following code shows how this could be done:

```
subroutine update(popsize);

   array objectives[1,1] /nosym;
   call dynamic_array(objectives, popsize, 3);

   array pareto[1] /nosym;
   call dynamic_array(pareto, popsize);

   array minmax[3] /nosym (1 -1 0);

   call GetSolutions(objectives, popsize, 2);
```

```
      call MarkPareto(pareto, npareto, objectives, minmax);

      do i = 1 to popsize;
         objectives[i,3] = pareto[i];
      end;

      call UpdateSolutions(objectives, popsize, 2);

      call SetElite(npareto);

   endsub;
```

This is an example of a user update routine that might be used in a multiobjective optimization problem. It is assumed that a user objective function has calculated two different objectives, and placed their values in the first two elements of segment 2 of the problem encoding. The first objective is to be maximized, and the second is to be minimized. Segment 2 has three elements, and the third element is used to mark the Pareto-optimal solutions. After dynamically allocating the necessary arrays from the *popsize* (population size) parameter, the update routine first retrieves the current solutions into the *objectives* array with the GetSolutions call. It then passes the *objectives* array directly to the MarkPareto call to perform the Pareto-optimal evaluations. Note that the *minmax* array directs the MarkPareto call to maximize the first element, minimize the second, and ignore the third element. After the MarkPareto call, the update routine writes the results back to the third element of the *objectives* array, and writes the *objectives* array back to the solution population with the UpdateSolutions call. This marks the solutions that comprise the Pareto-optimal set. The update routine then sets the *elite* parameter equal to the number of Pareto-optimal solutions with the SetElite call. It is assumed that the user has provided a fitness comparison function (designated with a SetSel call) that always selects a Pareto-optimal solution over a non-Pareto optimal one, so the *elite* setting guarantees that all the Pareto-optimal solutions are retained from generation to generation. Example 1.3 on page 67 illustrates the use of the MarkPareto call.

Mut Calls

executes a genetic mutation operator from within a user subroutine

This call can take one of several forms:

call MutDelta(*selected, seg, delta, n* **);**

call MutUniform(*selected, seg, n* **);**

call MutSwap(*selected, seg, n* **);**

call MutInvert(*selected, seg* **);**

The inputs to the subroutine are as follows:

selected	is an array that specifies the solution to be mutated.
seg	is the desired segment of the solution to which the mutation should be applied.
delta	is a vector of delta values for each component of the solution, used only for the Delta mutation operator.
n	specifies the number of components within the solution that should be mutated for the Delta and Uniform operators, and the number of swaps that should be made for the Swap operator.

The Mut subroutines should only be called from within a user mutation subroutine. Normally, this would only be done when a segmented encoding is used, or you want to modify the action of the standard mutation operator. For simple one-segment encoding, you would normally use a SetMut call to set the mutation operator and you do not need to define your own subroutine. The precise action of these mutation operators is described in the "Mutation Operators" section beginning on page 50.

PackBits Call

writes bits to specified variables for boolean encoding

call PackBits(*array, start, width, value* **);**

The inputs to the PackBits subroutine are as follows:

array	is an array to which the value is to be assigned.
start	is the starting position for the bit assignments.
width	is the number of bits to assign.
value	is the value to be assigned to the bits. For a single bit, this should be 0 or 1.

The PackBits subroutine is normally called within a user genetic operator subroutine to assign bit values to a boolean encoding. The *start* parameter should range in value from 1 to the size of the boolean encoding, and the encoding size should be greater than or equal to *start* + *width* - 1. Bits not within the specified range are not changed. The following code, which might occur in a mutation subroutine, first reads in the selected solution segment into s with the ReadMember call and then assigns ones to the first and second bits of the solution with the PackBits call before writing it back out to the current generation.

```
array s[2];
call ReadMember(selected, seg, s);
...
/* intervening code */
...
call PackBits(s, 1, 2, 3);
call WriteMember(selected, seg, s);
```

Program Statements

This section lists the program statements used to initialize the model, code the objective function, and control the optimization process in the GA procedure. It documents the differences between program statements in the GA procedure and program statements in the DATA step. The syntax of program statements used in PROC GA is identical to that used in the FCMP procedure.

Most of the program statements that can be used in the SAS DATA step can also be used in the GA procedure. See the *SAS Language Guide* or base SAS documentation for a description of the SAS program statements.

variable **=** *expression***;**
variable **+** *expression***;**
arrayvar[*subscript*] **=** *expression***;**
ABORT;
CALL *subroutine-name* < (*parameter-1* <, ...*parameter-n* >) >**;**
DELETE;
DO *program-statements;* **END;**
DO *variable* **=** *expression* **TO** *expression* <**BY** *expression*>**;**
        program-statements; **END;**
DO WHILE *expression* **;**
        program-statements; **END;**
DO UNTIL *expression* **;**
        program-statements; **END;**
GOTO *statement_label* **;**
IF *expression* **THEN** *program-statement***;**
    <**ELSE** *program-statement*>**;**
PUT < *variable(s)*> <@ | @@> **;**
RETURN <(*expression*)>**;**
SELECT <(*select-expression*)>**;**
    WHEN-*1* (*expression-1* <...,*expression-n*>)*program-statement* **;**
    <**WHEN**-*n* (*expression-1* <...,*expression-n*>)*program-statement* **;**>
    <**OTHERWISE** *program-statement* **;**>
STOP;
SUBSTR(*variable, index, length* **) =** *expression***;**

For the most part, the SAS program statements work as they do in the SAS DATA step as documented in the *SAS Language Guide*. However, there are several differences that should be noted.

- The ABORT statement does not allow any arguments.
- The DO statement does not allow a character index variable. Thus

  ```
  do i = 1,2,3;
  ```

  is supported; however,

  ```
  do i = 'A','B','C';
  ```

  is not.

- The PUT statement, used mostly for program debugging in PROC GA, supports only some of the features of the DATA step PUT statement, and has some new features that the DATA step PUT statement does not:
  - The PROC GA PUT statement does not support line pointers, factored lists, iteration factors, overprinting, _INFILE_, the colon (:) format modifier, or "$".
  - The PROC GA PUT statement does support expressions, but the expression must be enclosed inside parentheses. For example, the following statement displays the square root of x: **put (sqrt(x));**
  - The PROC GA PUT statement allows an array name without subscripts. The statement **PUT A;** prints all the elements of array **A**. The statement **PUT A=;** prints the elements of array **A** with each value labeled with the name of the element variable.
  - The PROC GA PUT statement supports the print item _PDV_ to print a formatted listing of all variables in the program. For example, the following statement displays a more readable listing of the variables than the _all_ print item: **put _pdv_;**
- The WHEN and OTHERWISE statements allow more than one target statement. That is, DO/END groups are not necessary for multiple statement WHENs. For example, the following syntax is valid:

```
SELECT;
WHEN ( exp1 )   stmt1;
                stmt2;
WHEN ( exp2 )   stmt3;
                stmt4;
END;
```

ReadChild Call

reads a segment from a selected child solution into an array, within a user crossover operator

call ReadChild(*selected, seg, n, values* **);**

The inputs to the ReadChild subroutine are as follows:

selected	specifies the family (parents and children) obtained from the selection process.
seg	specifies the solution segment to be read.
n	specifies the child in the family from which to read the solution segment.
values	specifies an array to receive the solution elements.

The ReadChild call is used to obtain the solution values for manipulation within a user crossover operator subroutine. Normally it is only needed if you need to augment the action of a GA procedure-supplied crossover operator. You might need to make modifications to satisfy constraints, for example. The *selected* parameter is passed into the user subroutine by the GA procedure. The *seg* parameter is the desired segment of the solution to be obtained. Segments, which correspond to different encodings in the encoding string, are numbered, starting from 1 as the first segment. The parameter *n* should be 1 to get the first child, 2 for the second. The parameter *values* is an array, which should be dimensioned large enough to contain the segment's encoding. For example, the following subroutine illustrates how you could use the Read/WriteChild calls to modify offspring generated with a standard genetic operator:

```
call SetEncoding('R5');

subroutine cross(selected[*]);

/* generate offspring with arithmetic crossover operator */
call CrossArithmetic(selected, 1); /* here 1 refers to segment 1*/

array child1[5];
array child2[5];

/* get elements of first child solution */
call ReadChild(selected, 1, 1, child1);

/* get elements of second child solution values */
call ReadChild(selected, 1, 2, child2);

...
/* code to modify elements in child1 and child2 */
...

call WriteChild(selected,1,1,child1);
call WriteChild(selected,1,2,child2);
```

ReadCompare Call

reads a segment from a selected solution into an array, within a user fitness comparison subroutine

call ReadCompare(*selected, seg, n, values* **);**

The inputs to the ReadCompare subroutine are as follows:

selected	specifies the pair of solutions to be compared, obtained from the selection process.
seg	specifies the solution segment to be read.
n	specifies the solution (1 or 2) from which to read the segment.
values	specifies an array to receive the solution elements.

The ReadCompare call is used to obtain the solution values for manipulation within a user fitness comparison subroutine, which can be designated in a SetSel call.

ReadMember Call

reads the selected solution into an array for a user objective function or mutation operator

call ReadMember(*selected, seg, destination* **);**

The inputs to the ReadMember subroutine are as follows:

selected	is a parameter passed to the user subroutine by the GA procedure, which points to the selected solution.
seg	specifies which segment of the solution to retrieve.
destination	specifies an array in which to store the solution elements.

The ReadMember call is used within a user objective function or mutation operator to obtain the elements of a selected solution and write them into a specified vector. They can then be used to compute an objective value, or in the case of a mutation operator, manipulated and written back out with a WriteMember call.

ReadParent Call

reads selected solution elements into an array in a user crossover subroutine

call ReadParent(*selected, seg, n, destination* **);**

The inputs to the ReadParent subroutine are as follows:

selected	is a parameter passed to the user subroutine by the GA procedure, which points to the selected solution family.
seg	is the segment of the desired parent solution to be obtained.
n	is the number of the parent, starting at 1.
destination	is an array in which to store the solution elements.

The ReadParent subroutine is called inside a user crossover operator subroutine to obtain the elements of selected parent solutions. Normally you would then manipulate and combine the elements of the two parents and use a WriteChild call to create the child offspring and complete the action of the crossover operator.

ReEvaluate Call

reruns the evaluation phase of the genetic algorithm

call ReEvaluate( );

The ReEvaluate call recomputes the objective values for the current generation. You do not normally need to use this call, because the GA procedure evaluates the objective function during the optimization process in the evaluation phase. This subroutine should be called from a user update or finalize routine if a parameter that affects the objective value or solution is changed. For example, you may have a user objective function that can perform an additional local optimization if a particular parameter is set. If your update routine changes that parameter, then you should call the ReEvaluate subroutine to update the solutions and objective function values.

SetBounds Call

sets constant upper and lower bounds

call SetBounds(*lower, upper* <, *seg*> **);**

The inputs to the SetBounds subroutine are as follows:

lower	is a lower bound for the solution components.
upper	is an upper bound for the solution components.
seg	is optional, and specifies a segment of the solution to which the bounds apply. If *seg* is not specified, then it defaults to a value of 1.

The SetBounds subroutine is used to establish upper and lower bounds on the solution space. It applies only to integer and real encoding. For multiple segment encoding, use the *seg* parameter to specify a segment other than the first. *upper* and *lower* must be arrays, with the same dimension as the encoding size. SetBounds must be called if you intend to specify default initialization for integer or real encoding and have not specified the FIRSTGEN= option, or if you use the Uniform mutation operator. The action of the standard mutation and crossover operators supplied by the GA procedure is automatically modified so that the bounds established by a SetBounds call are respected.

SetCross Calls

sets the crossover operator

This call can take one of several forms:

call SetCrossRoutine(*'rname'* **);**

call SetCrossSimple(*alpha* **);**

call SetCross2Point(*alpha* **);**

call SetCrossUniform(*alpha* **);**

call SetCrossArithmetic( );

call SetCrossHeuristic( );

call SetCrossOrder( );

call SetCrossPMatch( );

call SetCrossCycle( );

The inputs to the subroutines are as follows:

rname	is the name of a user subroutine to use for the crossover operator.
alpha	is a numeric parameter with a value between 0 and 1.

The SetCross subroutines are used to set the crossover operator for the genetic algorithm optimization process.

The SetCrossRoutine specifies that a user subroutine *rname* should be used. This parameter must be a literal quoted string; it cannot be a variable. See the "Defining User Genetic Operators" section on page 52 for information on defining a crossover operator.

The Simple, 2point, and Uniform operators are applicable to real, integer, and boolean encodings. The *alpha* parameter is used with real and integer encodings, and should be set to 1 for boolean. The actions of these operators are explained in detail in the "Crossover Operators" section beginning on page 44.

The Arithmetic operator is applicable to real and integer encoding, and combines parent solutions by interpolating between them. The advantage of this operator is that it automatically respects convex solution domains.

The Heuristic operator is applicable to real encoding. It is based on interpolation and projection using objective function values.

The Order, PMatch, and Cycle operators are applicable to sequence encoding. For the PMatch and Cycle operators, the absolute position of elements is emphasized. For the Order operator, the relative position of elements to each other is most important, and for that reason it is most often used for circular routing problems, such as the Traveling Salesman Problem.

All of the operators are explained and discussed in more detail in the "Crossover Operators" section beginning on page 44.

SetCrossProb Call

sets the crossover probability

call SetCrossProb(*p* **);**

The input to the SetCrossProb subroutine is as follows:

p	is the crossover probability.

The SetCrossProb subroutine is used to set the crossover probability for the genetic algorithm optimization process. The crossover probability p should be between 0 and 1. Typical values for this parameter range from 0.6 to 1.0. The crossover probability will be overridden if required by a SetElite call. The elite solutions are passed on to the next generation without undergoing crossover, regardless of the crossover probability.

SetElite Call

sets the number of best solutions to pass to the next generation

call SetElite(*elite* **);**

The input to the SetElite subroutine is as follows:

elite	is the number of best solutions to be passed unmodified from the current solution generation to the next.

The SetElite subroutine is used to ensure that the best solutions encountered in the optimization are not lost by the random selection process. In pure tournament selection, although better solutions are more likely to be selected, it is also possible that any given solution will not be chosen to participate in a tournament, and even if it is selected, it might be modified by crossover or mutation. The SetElite call modifies the optimization process such that the best *elite* solutions in the current population are exactly preserved and passed on to the next generation. This behavior is observed regardless of the crossover or mutation settings. When a SetElite call is made, the first *elite* solutions in the population retrieved by a GetSolutions call or output to a data set are the fittest, and these *elite* solutions are sorted so that the most fit is first. In general, using the SetElite call speeds the convergence of the optimization process. However, it can also lead to premature convergence before a true global optimum is reached. If no SetElite call is made, a default *elite* value of 1 is used by the GA procedure to make sure that the best solution encountered in the optimization process is never lost.

SetEncoding Call

specifies the problem encoding

call SetEncoding(*encoding* **);**

The input to the SetEncoding subroutine is as follows:

encoding	is a string used to specify the form of the solution.

The SetEncoding subroutine is used to establish the type of problem solution encoding. The *encoding* parameter should be a string of letter-number pairs, where the letter determines the type of encoding: I for integer, R for real-valued, S for sequences, and B for boolean bit strings. Each letter is followed by a number to indicate the number of components for that encoding. Multiple letter-number pairs can be used to specify a multisegment encoding. For example,

```
call SetEncoding('I10');
```

specifies that solutions are in the form of a 10-member integer vector, and

```
call SetEncoding('I5R10');
```

specifies that solutions have a 5-component integer segment and 10-component real-valued segment. See the "Using Multisegment Encoding" section on page 43 for details on multisegment encoding.

SetFinalize Call

designates a user subroutine to perform post-processing at the end of the optimization process

call SetFinalize(*'routine'* **);**

The input to the SetFinalize subroutine is as follows:

routine	is the name of a subroutine you have defined, which is called when the optimization process ends. This parameter must be a string literal; a variable is not accepted.

The SetFinalize subroutine enables you to define a subroutine to be called at the end of the optimization process. You might use this subroutine to perform additional refinements of the best solution, or you could generate and write out additional data for plots or reports.

SetMut Calls

sets the mutation operator

This call can take one of several forms:

call SetMutRoutine(*'rname'* **);**

call SetMutDelta(*delta, n* **);**

call SetMutUniform(*n* **);**

call SetMutSwap(*n* **);**

call SetMutInvert( );

The inputs to the SetMut subroutines are as follows:

rname	is the name of a user subroutine to use for the mutation operator.
delta	is a vector of perturbation values, one for each component in the encoding.
n	is the number of components to be perturbed.

The SetMut subroutines are used to set the type and parameters of the genetic mutation operator.

The SetMutRoutine specifies that a user subroutine *rname* should be used. This parameter must be a literal quoted string; it cannot be a variable. See the "Defining User Genetic Operators" section on page 52 for information on defining a mutation operator.

The Delta operator is applicable to integer and real encoding only. The operator randomly chooses *n* components of the solution, and adds or subtracts the corresponding *delta* value from it. The result is also truncated to fit within bounds specified in a SetBounds call.

The Uniform operator is applicable to integer, real, and boolean encoding. The operator randomly chooses *n* components of the solution. For boolean encoding, it then flips the corresponding bits. For integer or real encoding, it changes the component to a random value between the upper and lower bounds set by a SetBounds call.

The Swap and Invert operators are applicable only to sequence encoding. The Swap operator swaps the order of *n* pairs of components. The Invert operator, which is primarily used for the Traveling Salesman Problem, swaps one pair of components, and reverses the order of the components between them.

If no SetMut call is made, the GA procedure uses a default operator appropriate to the problem encoding. For real, integer, and boolean encoding, a Uniform operator with $n = 1$ is the default. For sequence encoding, the default is the Swap operator with $n = 1$. If the objective function has been set with a SetObjTSP call, the default is the Invert operator.

SetMutProb Call

sets the mutation probability

call SetMutProb(*p* **);**

The input to the SetMutProb subroutine is as follows:

p	is the mutation probability.

The SetMutProb subroutine is used to set the mutation probability for the genetic algorithm optimization. The probability *p* should be a number between 0 and 1, and is interpreted as the probability that a solution in the next generation should have the mutation operator applied to it. If a SetElite call has been made, then the elite solutions do not undergo mutation. Generally, a high mutation probability degrades the convergence of the genetic algorithm optimization, but some level of mutation is required to assure a thorough search and avoid premature convergence before the global optimum is found. Typical values for *p* are near 0.05 or less.

SetObj Call

sets the objective to be optimized

This call can take one of two forms:

call SetObjFunc(*'fname', minmax* **);**

call SetObjTSP(*distance* **);**

The inputs to the SetObj subroutine are as follows:

fname	is the name of a user objective function.
minmax	is set to 0 to minimize the objective, 1 to maximize.
distance	is a matrix of distances between locations for a Traveling Salesman Problem.

The SetObj subroutine is used to establish the objective function for the optimization process. The SetObjFunc call specifies that a user function is used for the objective function. The SetObjFunc call only accepts a literal string for the function name; you cannot use a variable or expression. See the "Defining an Objective Function" section on page 55 for more information on defining your own objective function.

The SetObjTSP call provides a quick way to specify the objective function for the Traveling Salesman Problem. The *distance* parameter is two-dimensional, and is a matrix of distances between locations, such that *distance*$[i,j]$ is the distance between locations i and j.

SetSel Call

sets the selection parameters

This call can take one of two forms:

call SetSelTournament(*size, <'fname'>* **);**

call SetSelDuel(*prob, <'fname'>* **);**

The inputs to the SetSel subroutine are as follows:

size	is the size of a tournament to be used to select a solution.
prob	is the probability of the fittest solution being selected in a tournament of size 2.
fname	is optional, representing the name of a user function that is to be used to compare the fitness of two solutions.

The SetSel call is used to set up the selection process, which selects members of the current generation to be propagated to the next. Selection is based on solution fitness, with the fittest solutions more likely to be selected. The fitness of a solution is generally determined by its objective value, with better objective values corresponding to higher fitness. However, there are techniques for multiple objective optimization and constraint handling that use more complicated criteria, and the optional *fname* parameter enables you to supply a fitness comparison routine to implement those techniques. For a simple single-objective problem, you do not have to supply this parameter; the objective value is used by default.

The SetSelTournament call is used to specify a tournament in which *size* solutions are chosen at random from the population, and the solution with the highest fitness is selected from that group. The higher the value of *size*, the greater the selection pressure. See the "Controlling the Selection Process" section on page 13 for guidance on setting this parameter.

The SetSelDuel call is used to specify a tournament of size 2, in which the fittest member is selected with the probability specified in *prob*. The parameter *prob* can be valued from 0.5 (corresponding to random selection) to 1.0. The selective pressure set up by this call is less than the selective pressure that can be set by the SetSelTournament call. A SetSelDuel call with $prob = 1$ is equivalent to a SetSelTournament call with $size = 2$.

In general, a higher selection pressure speeds the convergence of the genetic algorithm. However, it also increases the risk of premature convergence before the global optimum is reached. It is usually better to start the optimization process at a lower selective pressure, and increase it as the optimization continues. A SetSel can be made from inside a user update routine (designated with a SetUpdateRoutine call) to modify the selection parameters or method as the optimization progresses.

If you do not make a SetSel call, then the GA procedure uses a conventional tournament with $size = 2$.

SetUpdateRoutine Call

designates a control subroutine to be called at each iteration

call SetUpdateRoutine(*'routine'* **);**

The input to the SetUpdateRouting subroutine is as follows:

routine	is the name of a subroutine you have defined that is called once during each iteration of the optimization process. This parameter must be a string literal; a variable is not accepted.

The SetUpdate subroutine enables you to define a subroutine to be called at each iteration of the optimization process, in order to monitor the progress of the genetic algorithm, adjust optimization parameters, or perform calculations that depend on the population as a whole. The specified routine is called once at each iteration, just before the selection process, and after the evaluation phase. See the "Monitoring Progress and Reporting Results" section on page 17 for a discussion of how an update routine might be used.

UnpackBits Function

retrieves bit values from a packed bit array

r = UnpackBits(*source, start, width* **);**

The inputs to the UnpackBits function are as follows:

source	is an array containing the packed bit values.
start	is the starting bit, with the lowest bit starting at 1.
width	is the number of bits to retrieve. A value of 1 retrieves a single bit.

The UnpackBits function is invoked inside user objective and genetic operators to retrieve values of individual bits for boolean encoding. Normally *source* will have been filled by a ReadMember, ReadParent, or ReadChild call. The *start* parameter can range in value from 1 up to the size of the boolean encoding. The *width* parameter can range from 1, to get a single bit value (0 or 1), to the number of bits in the natural integer (32 on most computers in use today).

UpdateSolutions Call

updates current solution population

call UpdateSolutions(*sol, n, seg* **);**

The inputs to the UpdateSolutions subroutine are as follows:

sol	is an array containing the replacement solution elements.
n	is the number of solutions to update.
seg	is the segment of the solution to replace.

The UpdateSolutions subroutine is used to replace the values of the selected solution segment with new values computed in an update routine. The update routine can be designated in a SetUpdateRoutine call. The UpdateSolutions call is often used to implement advanced strategies such as marking Pareto-optimal sets or employing local optimizations. The *sol* parameter should have 2 dimensions. The first dimension represents the solution number, and should have a value of *n* or greater. The second dimension represents the element within the solution *seg*, and should be equal to the segment size.

WriteChild Call

assigns values to a selected child solution from within a user crossover operator

call WriteChild(*selected, seg, n, source* **);**

The inputs to the WriteChild subroutine are as follows:

selected	is an array specifying the selected family of solutions. The *selected* array is normally passed into the user subroutine that calls WriteChild, and should be passed unaltered to WriteChild.
seg	is the segment to which the elements are to be written.
n	is the child within the family to which the elements are to be written. A value of 1 is for the first child, 2 for the second, and so on.
source	is an array containing the values to be written.

The WriteChild subroutine is called inside a user crossover operator subroutine to assign to the elements of a selected child solution. It is normally used to complete the action of the crossover operator.

WriteMember Call

assigns values to a selected solution from within a user objective function or mutation operator

call WriteMember(*selected, seg, source* **);**

The inputs to the WriteMember subroutine are as follows:

selected	is an array specifying the selected family of solutions. The *selected* array is normally passed into the user subroutine that calls WriteMember, and should be passed unaltered to WriteMember.
seg	is the segment to which the elements are to be written.
source	is an array containing the values to be written.

The WriteMember subroutine is called inside a user objective function or mutation operator subroutine to assign values to the elements of a selected solution. It is normally used to complete the action of the objective function or mutation operator.

Details

Using Multisegment Encoding

The GA procedure enables you to represent problems with solutions consisting of mixed parameter types using multisegment encoding. Solutions can contain multiple segments, where each segment is a vector of one particular parameter type. Multiple segments can also be used to store additional information you want to keep for each solution for user objective functions or genetic operators. The utility functions provided by the GA procedure give you full access to read from and write to individual solution segments you define.

Segments are set up with a SetEncoding call. The input parameter to this call is a string consisting of letter-number pairs, with each pair describing the type and number of elements in one segment. The allowed letters and corresponding encodings are as follows:

R or *r*	specifies real encoding. The elements of the solution segment are real numbers. One common problem where this encoding is used is nonlinear function optimization over the real domain.
I or *i*	specifies integer encoding. The elements of the solution segment are integers. Examples of where this encoding might be used include assignment problems where the integers represent which resources are assigned to particular tasks, or in problems involving real variables that are constrained to be integers.
B or *b*	specifies boolean encoding. The elements of the solution consist of binary (0 or 1) bits. This type of encoding might be used, for example, to represent variables in a variable selection problem, or inclusion of particular items in a 0/1 Knapsack Problem.
S or *s*	specifies sequence encoding. The segment consists of randomly ordered sequences of integers ranging from 1 to the number of elements. For example, [2, 4, 5, 1, 3] is an example of S5 encoding, as is [5, 3, 2, 1, 4]. Sequence encoding is a natural way to represent routing optimizations like the Traveling Salesman Problem, or any problem optimizing permutations of parameters.

Suppose the problem is to optimize the scheduling of 20 tasks, and for each task you need to choose one machine out of a set of appropriate machines for each task. The natural encoding for that problem could be set up with

```
call SetEncoding('I20S20');
```

This specifies a two-segment solution encoding, with segment 1 (I20) an integer vector representing the machine assignment for each task, and segment 2 (S20) representing the sequence of tasks.

When you use multisegment encoding, you must also define subroutines to calculate the objective function and perform the genetic crossover and mutation operations. Within your subroutines, you can use utility functions provided by the GA procedure to extract and write out values to individual segments of the solution, and routines provided by the GA procedure to perform standard genetic crossover and mutation operations on selected segments. See the "Using Standard Genetic Operators" section beginning on page 44 for a discussion of the operators provided by the GA procedure, and the "Defining User Genetic Operators" section on page 52 and the "Defining an Objective Function" section on page 55 for details of defining user routines.

Using Standard Genetic Operators

The GA procedure includes a set of standard crossover and mutation operators, and also enables you to define your own genetic operator with a subroutine. The following sections describe the standard genetic operators and how to invoke them.

Crossover Operators

simple: This operator is defined for integer, real, and boolean encoding. This operator performs the following action: a position k within an encoding of length n is chosen at random, such that $1 \leq k < n$. Then for parents P and Q, the offspring are

$$\textit{child1} = [P_1, P_2, ..., P_k, Q_{k+1}, Q_{k+2}, ..., Q_n]$$

$$\textit{child2} = [Q_1, Q_2, ..., Q_k, P_{k+1}, P_{k+2}, ..., P_n]$$

For integer and real encoding, you can specify an additional parameter, a, where $0 < a \leq 1$. It modifies the offspring as follows:

$$p_i = aP_i + (1-a)Q_i, \; i = k+1, k+2, ..., n$$

$$q_i = aQ_i + (1-a)P_i, \; i = k+1, k+2, ..., n$$

$$\textit{child1} = [P_1, P_2, ..., P_k, q_{k+1}, q_{k+2}, ..., q_n]$$

$$\textit{child2} = [Q_1, Q_2, ..., Q_k, p_{k+1}, p_{k+2}, ..., p_n]$$

For integer encoding, the elements are then rounded to the nearest integer. For boolean encoding, the a parameter is ignored, and is effectively 1.

For single-segment encoding, you can specify the use of this operator with the call

```
call SetCrossSimple(a);
```

From within a user crossover subroutine, you can use

```
call CrossSimple(selected, seg, a);
```

where *selected* is the selection passed to your subroutine, and *seg* is the segment to which the simple crossover operator is to be applied.

2point: This operator is defined for integer, real, and boolean encoding of length $n \geq 3$. Two positions, $k1$ and $k2$, are chosen at random, such that $1 \leq k1 < k2 < n$. Element values between those positions are swapped between parents. For parents Q and P, the offspring are

$$\mathit{child1} = [P_1, P_2, ..., P_{k1}, Q_{k1+1}, ..., Q_{k2}, P_{k2+1}, ..., P_n]$$

$$\mathit{child2} = [Q_1, Q_2, ..., Q_{k1}, P_{k1+1}, ..., P_{k2}, Q_{k2+1}, ..., Q_n]$$

For real and integer encodings, you can specify an additional parameter, a, where $0 < a \leq 1$. It modifies the offspring as follows:

$$p_i = aP_i + (1-a)Q_i, \; i = k1+1, k1+2, ..., k2$$

$$q_i = aQ_i + (1-a)P_i, \; i = k1+1, k1+2, ..., k2$$

$$\mathit{child1} = [P_1, P_2, ..., P_{k1}, q_{k1+1}, ..., q_{k2}, P_{k2+1}, ..., P_n]$$

$$\mathit{child2} = [Q_1, Q_2, ..., Q_{k1}, p_{k1+1}, ..., p_{k2}, Q_{k2+1}, ..., Q_n]$$

Note that small values of a reduce the difference between the offspring and parents. For boolean encoding, a is always 1. For single-segment encoding, you can specify the use of this operator with the call

```
call SetCross2Point(a);
```

From within a user crossover subroutine, you can call

```
call Cross2Point(selected, seg, a);
```

where *selected* is the selection passed to your subroutine, and *seg* is the segment to which the two-point crossover operator is to be applied.

uniform: This operator is defined for integer, real, and boolean encoding of length $n \geq 3$. For parents Q and P, offspring q and p are generated such that

$$q_i = \begin{cases} Q_i, & \text{with probability 0.5} \\ aP_i + (1-a)Q_i, & \text{where } 0 \leq a \leq 1 \end{cases}$$

$$p_i = \begin{cases} P_i, & \text{with probability 0.5} \\ aQ_i + (1-a)P_i, & \text{where } 0 \leq a \leq 1 \end{cases}$$

Note that a determines how much interchange there is between parents. A low value of a implies little change between offspring and parents. For boolean encoding, a is always taken to be 1.

For single-segment encoding, you can specify the use of this operator with the call

`call SetCrossUniform(`*a***`);`**

From within a user crossover subroutine, you can call

`call CrossUniform(`*selected, seg, a***`);`**

where *selected* is the selection parameter passed to your crossover subroutine, and *seg* is the segment to which the uniform crossover operator is to be applied. In both cases, if the encoding is boolean, the a parameter is ignored, and a is effectively 1.

arithmetic: This operator is defined for real and integer encoding. It treats the solution segment as a vector, and computes offspring of parents P and Q as

$$\mathit{child1} = aP + (1 - a)Q$$

$$\mathit{child2} = aQ + (1 - a)P$$

where a is a random number between 0 and 1 generated by the GA procedure. For integer encoding, each component is rounded off to the nearest integer. It has the advantage that it always produces feasible offspring for a convex solution space. A disadvantage of this operator is that it tends to produce offspring toward the interior of the search region, so that it may not work if the optimum lies on or near the search region boundary. For single-segment encoding, you can specify the use of this operator with the call

`call SetCrossArithmetic();`

From within a user crossover subroutine, you can call

`call CrossArithmetic(`*selected, seg***`);`**

where *selected* is the selection parameter passed to your subroutine, and *seg* is the segment to which the arithmetic crossover operator is to be applied.

heuristic: This operator is defined for real encoding. It treats the solution segments as real vectors. It computes the first offspring from two parents P and Q, where Q is the parent with the best objective value, as

$$\mathit{child1} = a(Q - P) + Q$$

$$\mathit{child2} = aQ + (1 - a)P$$

where a is a random number between 0 and 1 generated by the GA procedure. The first child is a projection, and the second child is a convex combination, as with the arithmetic operator. This operator is unusual in that it uses the objective value. It has the advantage of directing the search in a promising direction, and automatically fine-tuning the search in an area where solutions are clustered. If the solution space has upper and lower bound constraints, the offspring are checked against the bounds, and any component outside its bound is set equal to that bound. The heuristic operator performs best when the objective function is smooth, and may not work well if the objective function or its first derivative is discontinuous. For single-segment encoding, you can specify the use of this operator with the call

```
call SetCrossHeuristic();
```

From within a user crossover subroutine, you can call

`call CrossHeuristic(`*selected*, *seg***`);`**

where *selected* is the selection parameter passed to your subroutine, and *seg* is the segment to which the heuristic crossover operator is to be applied.

pmatch: The partial match operator is defined for sequence encoding. It produces offspring by transferring a subsequence from one parent, and filling the remaining positions in a way consistent with the position and ordering in the other parent. Start with two parents and randomly chosen cutpoints as indicated:

$$P = [1, 2, |3, 4, 5, 6, |7, 8, 9]$$

$$Q = [8, 7, |9, 3, 4, 1, |2, 5, 6]$$

The first step is to cross the selected subsegments (note that '.' indicates positions yet to be determined):

$$\textit{child1} = [., ., 9, 3, 4, 1, ., ., .]$$

$$\textit{child2} = [., ., 3, 4, 5, 6, ., ., .]$$

Next, define a mapping according to the two selected subsegments:

```
9-3, 3-4, 4-5, 1-6
```

Then, fill in the positions where there is no conflict from the corresponding parent:

$$\textit{child1} = [., 2, 9, 3, 4, 1, 7, 8, .]$$

$$\textit{child2} = [8, 7, 3, 4, 5, 6, 2, ., .]$$

Last, fill in the remaining positions from the subsequence mapping. In this case, for the first child, $1 \rightarrow 6$ and $9 \rightarrow 3$, $3 \rightarrow 4$, $4 \rightarrow 5$, and for the second child, $5 \rightarrow 4$, $4 \rightarrow 3$, $3 \rightarrow 9$ and $6 \rightarrow 1$.

$$\mathit{child1} = [6, 2, 9, 3, 4, 1, 7, 8, 5]$$

$$\mathit{child2} = [8, 7, 3, 4, 5, 6, 2, 9, 1]$$

This operator tends to maintain similarity of both the absolute position and relative ordering of the sequence elements, and is useful for a wide range of sequencing problems. For single-segment encoding, you can specify the use of this operator with the call

`call SetCrossPMatch();`

From within a user crossover subroutine, you can call

`call CrossPMatch(`*selected*, *seg***`);`**

where *selected* is the selection parameter passed to your subroutine, and *seg* is the segment to which the partial match crossover operator is to be applied.

order: This operator is defined for sequence encoding. It produces offspring by transferring a randomly chosen subsequence of random length and position from one parent, and filling the remaining positions according to the order from the other parent. For parents P and Q, first choose two random cutpoints to define a subsequence:

$$P = [1, 2, |3, 4, 5, 6, |7, 8, 9]$$

$$Q = [8, 7, |9, 3, 4, 1, |2, 5, 6]$$

$$\mathit{child1} = [., ., 3, 4, 5, 6, ., ., .]$$

$$\mathit{child2} = [., ., 9, 3, 4, 1, ., ., .]$$

Starting at the second cutpoint, the elements of Q in order are (cycling back to the beginning):

```
2 5 6 8 7 9 3 4 1
```

after removing 3, 4, 5 and 6, which have already been placed in $\mathit{child1}$, we have:

```
2 8 7 9 1
```

Placing these back in order starting at the second cutpoint yields

$$\mathit{child1} = [9, 1, 3, 4, 5, 6, 2, 8, 7]$$

Applying this logic to $\mathit{child2}$ yields

$$\mathit{child2} = [5, 6, 9, 3, 4, 1, 7, 8, 2]$$

This operator maintains the similarity of the relative order, or adjacency, of the sequence elements of the parents. It is especially effective for circular path-oriented optimizations, such as the Traveling Salesman Problem. For single-segment encoding, you can specify the use of this operator with the call

```
call SetCrossOrder();
```

From within a user crossover subroutine, you can call

call CrossOrder(*selected*, *seg***);**

where *selected* is the selection parameter passed to your subroutine, and *seg* is the segment to which the order crossover operator is to be applied.

cycle: This operator is defined for sequence encoding. It produces offspring such that the position of each element value in the offspring comes from one of the parents. For example, consider parents P and Q,

$$P = [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

$$Q = [8, 7, 9, 3, 4, 1, 2, 5, 6]$$

For the first child, pick the first element from the first parent:

$$\mathit{child1} = [1, ., ., ., ., ., ., ., .]$$

To maintain the condition that the position of each element value must come from one of the parents, the position of the ‘8’ value must come from P, because the ‘8’ position in Q is already taken by the ‘1’ in $\mathit{child1}$:

$$\mathit{child1} = [1, ., ., ., ., ., ., 8, .]$$

Now the position of ‘5’ must come from P, and so on until the process returns to the first position:

$$\mathit{child1} = [1, ., 3, 4, 5, 6, ., 8, 9]$$

At this point, choose the remaining element positions from Q:

$$\mathit{child1} = [1, 7, 3, 4, 5, 6, 2, 8, 9]$$

For the second child, starting with the first element from the second parent, similar logic produces

$$\mathit{child2} = [8, 2, 9, 3, 4, 1, 7, 5, 6]$$

This operator is most useful when the absolute position of the elements is of most importance to the objective value. For single-segment encoding, you can specify the use of this operator with the call

```
call SetCrossCycle();
```

From within a user crossover subroutine, you can call

call CrossCycle(*selected, seg***);**

where *selected* is the selection parameter passed to your subroutine, and *seg* is the segment to which the cycle crossover operator is to be applied.

Mutation Operators

uniform: This operator is defined for real or integer encoding with upper and lower bounds specified with a SetBounds call. To apply this operator, a position k is randomly chosen within the solution S, and S_k is modified to a random value between the upper and lower bounds for element k. The process is repeated n times. For single-segment encoding, you can specify this operator with the call

call SetMutUniform(*n***);**

For multisegment encoding, you can invoke this operator on a solution segment from within a user mutation subroutine with

call MutUniform(*selected, segment, n***);**

where *selected* is the selection parameter passed into the subroutine.

This operator may prove especially useful in early stages of the optimization, since it tends to distribute solutions widely across the search space, and to avoid premature convergence to a local optimum. However, in later stages of an optimization, when the search needs to be fine-tuned to hone in on an optimum, the uniform operator may hinder the optimization.

delta: This operator is defined for integer and real encoding. It first chooses n elements of the solution at random, and then perturbs each element by a fixed amount, set by a *delta* input parameter. The *delta* parameter is an array with the same length as the encoding. A randomly chosen element k of the solution S is modified such that

$$S_k \in \{S_k - \mathit{delta}_k, S_k + \mathit{delta}_k\}$$

If upper and lower bounds are specified with a SetBounds call, then S_k is adjusted as necessary to fit within the bounds. This operator gives you the ability to fine-tune the search by modifying the magnitude of the *delta* vector. One possible strategy is to start with larger *delta* values, and then reduce them as the search progresses and begins to converge to an optimum. This operator is also useful if the optimum is known to be on or near a boundary, in which case *delta* can be set large enough to always perturb the solution element to a boundary. For single-segment encoding, you can specify this operator with the call

call SetMutDelta(*delta*, *n***);**

For multisegment encoding, you can invoke this operator on a solution segment from within a user mutation subroutine with

call MutDelta(*selected*, *segment*, *delta*, *n***);**

where *selected* is the selection parameter passed into the subroutine.

swap: This operator is defined for sequence problem encoding. It picks two random locations in the solution vector and swaps their values. You can also specify that multiple swaps be made for each mutation. For single-segment encoding, you can specify the swap operator with the call

call SetMutSwap(*n***);**

where n is the number of swaps for each mutation. For multisegment encoding, you can invoke this operator on a solution segment from within a user mutation subroutine with

call MutSwap(*selected*, *segment*, *n***);**

where *selected* is the selection parameter passed into the subroutine.

invert: This operator is defined for sequence encoding. It picks two locations at random and reverses the order of elements between them. This operator is most often applied to the Traveling Salesman Problem. For single-segment encoding, you can specify this operator with the call

call SetMutInvert();

For multisegment encoding, you can invoke this operator on a solution segment from within a user mutation subroutine with

call MutInvert(*selected*, *segment***);**

where *selected* is the selection parameter passed into the subroutine.

The standard crossover and mutation operators that are allowed for each encoding type are summarized in Table 1.1.

Table 1.1. Valid Genetic Operators for Each Encoding

Encoding	**Crossover**	**Mutation**
real	user subroutine simple 2point arithmetic heuristic	user subroutine uniform delta
integer	user subroutine simple 2point arithmetic	user subroutine uniform delta
boolean	user subroutine simple 2point uniform	user subroutine uniform
sequence	user subroutine pmatch order cycle	user subroutine swap invert

Defining User Genetic Operators

You can define new genetic operators with subroutines. The GA procedure calls your subroutine when it is necessary to perform mutation or crossover operations. You can designate that a subroutine be used for crossover with the call

```
call SetCrossRoutine('name');
```

where *name* is the name of the subroutine you have defined. Similarly, you can designate a subroutine for the mutation operator with

```
call SetMutRoutine('name');
```

The subroutine name must be a quoted string. The first parameter of the crossover or mutation subroutine you define must be a numeric array. When the GA procedure calls your subroutine, it passes information in the first parameter, referred to as the *selection* parameter, which designates the selected members for the operation. You should not alter the selection parameter in any way, but pass it unchanged into special utility routines provided by the GA procedure in order to obtain the solution elements and write them out to the selected members. You can define as many other parameters to your subroutine as you need; they are filled in with values from variables of the same name created in the global portion of your program. Any array parameters must be numeric and of type /NOSYMBOLS. All parameters are passed by reference to your subroutine; therefore, changes to these parameters inside the subroutine are passed back to the calling environment, which makes it easy to update global parameters and tabulate cumulative data as the optimization progresses. However, you should be careful not to alter a parameter that you intend to remain constant.

For a crossover subroutine, use the ReadParent call to get the elements of the selected parents into arrays that you can then manipulate with programming statements. The results can be written out to the designated offspring with a WriteChild call. The following code is an example of a crossover subroutine. The subroutine creates two new offspring from two selected parents by switching the odd-numbered elements between the two parents.

```
/* single-segment integer encoding of size 10 */
call SetEncoding('I10');

/* encoding size is 10 */
n = 10;

subroutine swapodd(selected[*], n);
  array child1[1] /nosym;
  array child2[1] /nosym;

  /* reallocate child arrays to right size */
  call dynamic_array(child1,n);
  call dynamic_array(child2,n);

  /* read segment 1 from parent 1 into child1 */
  call ReadParent(selected, 1, 1, child1);

  /* read segment 1 from parent 2 into child2 */
  call ReadParent(selected, 1, 2, child2);

  /* swap the odd elements in the solution */
  do i = 1 to n by 2;
    temp = child1[i];
    child1[i] = child2[i];
    child2[i] = temp;
  end;

  /* write offspring out to selected children */
  call WriteChild(selected, 1, 1, child1);
  call WriteChild(selected, 1, 2, child2);
endsub;

/* designate swapodd as the crossover routine */
call SetCrossRoutine('swapodd');
```

The next sample code illustrates a crossover routine that might be used for multi-segment mixed integer and sequence encoding. The subroutine uses the standard Simple crossover operator for the integer segment, and the PMatch operator for the sequence-encoded segment.

```
/* Solution has 2 segments, integer I5 and sequence S5 */
call SetEncoding('I5S5');

/* alpha parameter for Simple crossover operator */
alpha = 1;

subroutine mixedIS(selected[*], alpha);
```

```
     /* execute simple operator on segment 1 */
     call CrossSimple(selected, 1, alpha);

     /* execute pmatch operator on segment 2 */
     call CrossPMatch(selected, 2);

   endsub;

   call SetCrossRoutine('mixedIS');
```

For a mutation subroutine, use a ReadMember call to obtain the elements of the solution selected to be mutated, and a WriteMember call to write the mutated elements back out to the solution. For example, the following statements define a mutation subroutine that swaps two adjacent elements at a randomly chosen position in a sequence.

```
   /* Solution has 1 segment, sequence S10 */
   call SetEncoding('S10');

   n = 10;

   subroutine swap2(selected[*], n);

     /* declare an array for working memory */
     array member[1] /nosym;

     /* allocate array to required length */
     call dynamic_array(member, n);

     /* read segment 1 of selected member into array */
     call ReadMember(selected,1,member);

     /* generate random number between 0 and 1 */
     r = rand('uniform');

     /* convert r to integer between 1 and n-1 */
     i = int(r * (n - 1)) + 1;

     /* swap element values */
     temp = member[i];
     member[i] = member[i+1];
     member[i+1] = temp;

     /* write result back out to solution */
     call WriteMember(selected,1,member);

   endsub;

   /* Set the mutation routine to swap2 */
   call SetMutRoutine('swap2');
```

Defining an Objective Function

The GA procedure enables you to specify your objective to be optimized with a function you create, or as a standard objective function that the GA procedure provides. Currently the only standard objective you can specify without writing an objective function is the Traveling Salesman Problem, which can be specified with a SetObjTSP call. In the future, other objective functions will be added. You can designate a user objective function with the call

```
call SetObjFunc('name', minmax);
```

where *name* is the name of the function you have defined, and *minmax* is set to 0 to specify a minimum or 1 to specify a maximum.

A user objective function must have a numeric array as its first parameter. When the GA procedure calls your function, it passes an array in the first parameter that specifies the selected solution, which is referred to as the *selection* parameter. The selection parameter must not be altered in any way by your function. Your function should pass the selection parameter to a ReadMember call to read the elements of the selected solution into an array. Your function can then access this array to compute an objective value, which it must return. As with the genetic operator routines, you can define additional arguments to your objective function, and the GA procedure passes in variables with corresponding names that you have created in your global program. For example, the following statements set up an objective function that minimizes the sum of the squares of the solution elements.

```
call SetEncoding('R5');

n = 5;

function sumsq(selected[*], n);

  /* set up a scratch array to hold solution elements */
  array x[1] /nosym;

  /* allocate x to hold all solution elements */
  call dynamic_array(x, n);

  /* read members of the selected solution into x */
  call ReadMember(selected, 1, x);

  /* compute the sum of the squares */
  sum = 0;
  do i = 1 to n;
    sq = x[i] * x[i];
    sum = sum + sq;
  end;

  /* return the objective value */
  return(sum);
endsub;

call SetObjFunc('sumsq', 0);
```

In this example, the function **sumsq** is defined, and the SetObjFunc call establishes it as the objective function. The 0 for the second parameter of the SetObjFunc call indicates that the objective should be minimized. Note that the second parameter to the **sumsq** function, *n*, is defined in the procedure, and the value assigned to it there is passed into the function.

Defining a User Initialization Routine

For problems with simple constant bounds or simple sequencing problems it is not necessary to define a user initialization subroutine; simply specify 'DEFAULT' in the Initialize call. Defining a routine is only necessary if you need to satisfy more complicated constraints or apply some initial heuristics or local optimizations. A user initialization routine is specified with an Initialize call:

call Initialize('*name*', *size***);**

where *name* is the name of your initialize routine. The first parameter of the subroutine you define must be a numeric array. When the GA procedure calls your subroutine, it passes information in the first parameter, referred to as the *selection* parameter, which designates the member selected for initialization. Your subroutine should generate one solution and write out the values of the solution elements with a WriteMember call, using the selection parameter passed to your subroutine. The random number functions from base SAS are available to your subroutine, if needed. You can define as many other parameters to your subroutine as you need; they are filled in with values from variables of the same name created in your global program. The array used to write the generated solution out to the population must be numeric and declared with the /NOSYMBOLS option, as well as any arrays passed as parameters into your subroutine.

The following sample statements illustrate how to define an initialization routine. The feasible region is a triangle with vertices (0,0), (0,1) and (1,1).

```
call SetEncoding('R2');

/* set vertices of triangle (0,0), (0,1), and (1,1) */
array vertex1[2] /nosym (0,0);
array vertex2[2] /nosym (0,1);
array vertex3[2] /nosym (1,1);


subroutine triangle(selected[*], vertex1[2], vertex2[2], vertex3[2]);

array x[2] /nosym;

/* select 3 random numbers 0 < r < 1 */
r1 = rand('uniform');
r2 = rand('uniform');
r3 = rand('uniform');

/* normalize so r1 + r2 + r3 = 1 */
sumr = r1 + r2 + r3;
```

```
r1 = r1 / sumr;
r2 = r2 / sumr;
r3 = r3 / sumr;

/* form a convex combination of vertices in x */
do i = 1 to 2;
  x[i] = r1 * vertex1[i] + r2 * vertex2[i] + r3 * vertex3[i];
end;

/* write x out to the selected population member, to segment 1 */
call WriteMember(selected, 1, x);
endsub;

[other programming statements]

call Initialize('triangle',100);
```

In this example, the triangle initialization subroutine generates a solution that is a random convex combination of three points, which places it in the interior of the triangular region defined by the points. Note the use of the base SAS **`rand()`** function to get random numbers uniformly distributed between 0 and 1. The random numbers are then normalized so that their sum is 1. In the loop, they are used to compute a convex linear combination of the vertices, and the WriteMember call writes the solution out to the selected population member. The encoding specified a single segment, so the WriteMember call specifies segment 1 as the target. When the GA procedure executes the Initialize call, it executes the triangle routine 100 times, once for each member of the initial population.

Incorporating Heuristics and Local Optimizations

It is often effective to combine the genetic algorithm technique and other local optimizations or heuristic improvements. This can be done within the GA procedure by incorporating a local optimization into a user objective function and returning an improved objective value. Your user objective function can either replace the original solution with the optimized one, or you can leave the solution unchanged, replacing it with the optimized one only at the final iteration.

Replacing the original solution with the locally optimized one speeds convergence, but it also increases the risk of converging prematurely. If you choose to do so, you can modify the solution by writing the changed solution back to the population with a WriteMember call. You could also consider replacing the original solution with some probability p. For some problems, values of p from 0.05 to 0.15 have been shown to significantly improve convergence while avoiding premature convergence to a local optimum. This technique is illustrated in Example 1.1 on page 61.

Handling Constraints

Practical optimization problems usually involve constraints, which may make the problem harder to solve. Constraints are handled in genetic algorithms in several ways.

Encoding Strategy

The simplest approach is to set the problem encoding, genetic operators, and initialization such that the constraints are automatically satisfied. Fixed constant bounds are easily handled in this manner in the GA procedure with the SetBounds call. The default initialization process and genetic operators provided by the GA procedure automatically respects bounds specified in this manner. For some types of constraints, you may be able to create a direct mapping from a constrained solution domain to a second domain with simple constant bounds. You could then define your genetic operator to map the solution into the second domain, apply one of the standard genetic operators, and then map the result back to the original domain.

If the problem contains equality constraints, you should try to transform the problem to eliminate the equality constraints and reduce the number of variables. This strategy is opposite from what is usually done in linear programming, where inequality constraints are turned into equality constraints by the addition of slack variables.

Repair Strategy

If the constraints are more complex and cannot be easily satisfied automatically by the genetic operators, you may be able to employ a repair strategy: check the solution and modify it to satisfy the constraints. The check and repair can be done in a user genetic operator when the solution is generated, or it can be done in the evaluation phase in a user objective function. Possible strategies for making a repair inside an objective function include projecting the solution onto the constraint boundary; while inside a genetic operator you might try adjusting an operator parameter until the constraint is satisfied. If you do the repair in the objective function, you should compute the objective value after performing the repair. You can write the repaired solution back out to the population with a WriteMember call from a user objective function or mutation subroutine, and with a WriteChild call from within a crossover subroutine. Example 1.2 on page 64 illustrates the use of the repair strategy.

Penalty Strategy

Another technique is to allow solutions to violate constraints, but to also impose a fitness penalty that causes the population to evolve toward satisfying constraints as well as optimizing the objective. One way of employing this strategy is to simply add a penalty term to the objective function, but this approach should be used with care, as it is not always obvious how to construct the penalty function in a way that does not bias the optimization of the desired objective.

Direct Comparison Strategy

Using tournament selection opens another possibility for handling constraints. Define a fitness comparison routine (designated in a SetSel call) that employs the following logic:

1: If neither solution is feasible, choose the one closest to satisfying the constraints.

2: If one solution is feasible, and the other is not, choose the feasible one.

3: If both solutions are feasible, choose the one with the best objective value.

This strategy has the advantage that the objective function does not have to be calculated for infeasible solutions. To implement this method, you need to provide a measure of constraint violation and compute it in a user objective function; this value can be used in the first comparison step outlined above. For linear constraints, the GA procedure provides the EvaluateLC call for this purpose. This technique is illustrated in Example 1.3 on page 67. The technique works best when the solution space normally contains a significant number of solutions that satisfy the constraints. Otherwise it is possible that a single feasible solution might quickly dominate the population. In such cases, a better approach might be the following Bicriteria Comparison Strategy.

Bicriteria Comparison Strategy

A variation of the direct comparison strategy that has proved effective in many applications is the multiobjective, bicriteria approach. This strategy involves adding a second objective function, which is the magnitude of the constraint violation. Based on the original and constraint violation objective functions, a Pareto-optimal set of solutions is evolved in the population, and the Pareto-optimal set is evolved toward 0 constraint violation. See “Optimizing Multiple Objectives” for a full discussion of Pareto optimality and how to apply this technique.

Optimizing Multiple Objectives

Many practical optimization problems involve more than one objective criteria, where the decision maker needs to examine tradeoffs between conflicting objectives. With traditional optimization methods, these problems are often handled by aggregating multiple objectives into a single scalar objective, usually accomplished by some linear weighting of the multiple criteria. Other approaches involve turning objectives into constraints. One disadvantage of this strategy is that many separate optimizations with different weighting factors or constraints need to be performed to examine the tradeoffs between different objectives. Genetic algorithms enable you to attack multiobjective problems directly, to evolve a set of solutions in one run of the optimization process instead of solving multiple separate problems.

This approach seeks to evolve the Pareto-optimal set: the set of solutions such that for each solution, all the objective criteria cannot be simultaneously improved. This is expressed mathematically by the concept of Pareto optimality. A Pareto-optimal set is the set of all nondominated solutions, according to the following definition for *dominated*:

For an n-objective minimizing optimization problem, for each objective function f_i, a solution p is *dominated by* q if

$$f_i(p) \geq f_i(q) \text{ for all } i = 1, \ldots, n \text{ and } f_j(p) > f_j(q) \text{ for some } j = 1, \ldots, n$$

The following is one strategy that can be employed in the GA procedure to evolve a set of Pareto-optimal solutions to a multiobjective optimization problem:

user objective function: Define and specify a user objective function in a SetObjFunc call that computes each of the objective criteria and stores the objective values in one single solution segment.

user update routine: Define and specify a user update routine in a SetUpdateRoutine call that examines the entire solution population and marks those in the Pareto-optimal set. This can be done with the MarkPareto call provided by the GA procedure. Also, set the *elite* parameter equal to the number of Pareto-optimal solutions found.

selection criteria: Define a fitness comparison routine that favors the least dominated solutions, and designate it in a SetSel call. For selecting between two solutions when neither solution dominates the other, your routine can check a secondary criterion to direct the search to the area of ultimate interest.

The multiple objective values are recorded in one segment to enable the use of the MarkPareto call provided by the GA procedure. Setting the *elite* selection parameter to the size of the Pareto-optimal set, in conjunction with the comparison criteria, guarantees that the Pareto-optimal set in each generation is preserved to the next.

The secondary comparison criterion can be used to ensure that the final Pareto-optimal set is distributed in the area of ultimate interest. For example, for the Bicriteria Constraint Strategy described previously, the actual area of interest is where there is zero constraint violation, which is the second objective. The secondary comparison criterion in that case is to minimize the value of the constraint violation objective. After enough iterations, the population should evolve to the point that the best solution to the bicriteria problem is also the best solution to the original constrained problem, and the other Pareto-optimal solutions can be examined to analyze the sensitivity of the optimum solution to the constraints. For other types of problems, you may need to implement a more complicated secondary comparison criterion to avoid “crowding” of solutions about some arbitrary point, and ensure the evolved Pareto-optimal set is distributed over a range of objective values.

Examples

Example 1.1. Traveling Salesman Problem with Local Optimization

This example illustrates the use of the GA procedure to solve a Traveling Salesman Problem (TSP); it combines a genetic algorithm with a local optimization strategy. The procedure finds the shortest tour of 20 locations randomly oriented on a two-dimensional x-y plane, where $0 \le x \le 1$ and $0 \le y \le 1$. The location coordinates are input in the following DATA step:

```
/* 20 random locations for a  Traveling Salesman Problem */
data locations;
   input x y;
   datalines;
0.0333692 0.9925079
0.6020896 0.0168807
0.1532083 0.7020444
0.3181124 0.1469288
0.1878440 0.8679120
0.9786112 0.4925364
0.7918010 0.7943144
0.5145329 0.0363478
0.5500754 0.8324617
0.3893757 0.6635483
0.9641841 0.6400201
0.7718126 0.5463923
0.7549037 0.4584584
0.2837881 0.7733415
0.3308411 0.1974851
0.7977221 0.1193149
0.3221207 0.7930478
0.9201035 0.1186234
0.2397964 0.1448552
0.3967470 0.6716172
;
```

First, the GA procedure is run with no local optimizations applied.

```
proc ga data1 = locations seed = 55555;
call SetEncoding('S20');
ncities = 20;
array distances[20,20] /nosym;
do i = 1 to 20;
  do j = 1 to i;
    distances[i,j] = sqrt((x[i] - x[j])**2 + (y[i] - y[j])**2);
    distances[j,i] = distances[i,j];
  end;
end;
call SetObjTSP(distances);
call SetCrossOrder();
call SetMutInvert();
call Initialize('DEFAULT',100);
call ContinueFor(220);
run;
```

The proc GA statement uses a DATA1= option to get the contents of the locations data set, which creates array variables x and y from the corresponding fields of the data set. A solution will be represented as a circular tour, modeled as a 20-element sequence of locations, which is set up with the SetEncoding call. The array variable *distances* is created, and the loop initializes *distances* from the x and y location coordinates such that *distances*$[i, j]$ is the Euclidean distance between location i and j. Next, the SetObjTSP call specifies that the GA procedure use the included TSP objective function with the *distances* array. Next, the genetic operators are specified, with SetCrossOrder for the crossover operator and SetMutInvert for the mutation operator. Since the crossover probability and mutation probability are not explicitly set in this example, the default values of 1 and 0.05, respectively, are used. The selection parameters are not explicitly set (with SetSel and SetElite calls), so by default, a tournament of size 2 is used, and an *elite* parameter of 1 is used. Next, the Initialize call specifies default initialization (random sequences) and a population size of 100. The ContinueFor call specifies a run of 220 iterations. This value is a result of experimentation, after it was determined that the solution did not improve with more iterations. The output of this run of PROC GA is given in Output 1.1.1.

Output 1.1.1. Simple Traveling Salesman Problem

```
                         PROC GA Optimum Values

                                Objective

                              3.7465311323


                               Solution
                        Element            Value

                              1               13
                              2               12
                              3                6
                              4               11
                              5                7
                              6                9
                              7               20
                              8               10
                              9               17
                             10               14
                             11                5
                             12                1
                             13                3
                             14               19
                             15                4
                             16               15
                             17                8
                             18                2
                             19               16
                             20               18
```

The following program illustrates how the problem can be solved in fewer iterations by employing a local optimization. Inside the user objective function, before computing the objective value, every adjacent pair of cities in the tour is checked to determine if reversing the pair order would improve the objective value. For a pair of locations S_i and S_{i+1}, this means comparing the distance traversed by the subsequence $\{S_{i-1}, S_i, S_{i+1}, S_{i+2}\}$ to the distance traversed by the subsequence $\{S_{i-1}, S_{i+1}, S_i, S_{i+2}\}$, with appropriate wrap-around at the endpoints of the sequence. If the distance for the swapped pair is smaller than the original pair, then the reversal is done, and the improved solution is written back to the population.

```
proc ga data1 = locations seed = 55555;
call SetEncoding('S20');
ncities = 20;
array distances[20,20] /nosym;
do i = 1 to 20;
  do j = 1 to i;
    distances[i,j] = sqrt((x[i] - x[j])**2 + (y[i] - y[j])**2);
    distances[j,i] = distances[i,j];
  end;
end;

/* Objective function with local optimization */
function TSPSwap(selected[*],ncities,distances[*,*]);

  array s[1] /nosym;
  call dynamic_array(s,ncities);
  call ReadMember(selected,1,s);

  /* First try to improve solution by swapping adjacent cities */
  do i = 1 to ncities;
     city1 = s[i];
     inext = 1 + mod(i,ncities);
     city2 = s[inext];
     if i=1 then
        before = s[ncities];
     else
        before = s[i-1];
     after = s[1 + mod(inext,ncities)];
     if (distances[before,city1]+distances[city2,after]) >
        (distances[before,city2]+distances[city1,after]) then do;
        s[i] = city2;
        s[inext] = city1;
     end;
  end;
  call WriteMember(selected,1,s);

  /* Now compute distance of tour */
  distance = distances[s[ncities],s[1]];
  do i = 1 to (ncities - 1);
     distance + distances[s[i],s[i+1]];
  end;
  return(distance);
endsub;
call SetObjFunc('TSPSwap',0);
call SetCrossOrder();
call SetMutInvert();
call Initialize('DEFAULT',100);
call ContinueFor(85);
run;
```

The output after 85 iterations is given in Output 1.1.2.

Output 1.1.2. Traveling Salesman Problem with Local Optimization

```
                              PROC GA Optimum Values

                                     Objective

                                   3.7465311323


                                     Solution
                             Element              Value

                                   1                  8
                                   2                  2
                                   3                 16
                                   4                 18
                                   5                 13
                                   6                 12
                                   7                  6
                                   8                 11
                                   9                  7
                                  10                  9
                                  11                 20
                                  12                 10
                                  13                 17
                                  14                 14
                                  15                  5
                                  16                  1
                                  17                  3
                                  18                 19
                                  19                  4
                                  20                 15
```

Since all tours are circular, the actual starting point does not matter and this solution is equivalent to that reached with the simple approach without local optimization. It is reached after only 85 iterations, versus 220 with the simple approach.

Example 1.2. Nonlinear Objective with Constraints Using Repair Mechanism

This example illustrates the use of a repair mechanism to satisfy problem constraints. The problem is to minimize the six-hump camel-back function (Michalewicz 1996, Appendix B):

$$f(x) = \left(4 - 2.1x_1^2 + \frac{x_1^4}{3}\right) x_1^2 + x_1x_2 + \left(-4 + 4x_2^2\right) x_2^2$$

where x is within the triangle with vertices $V_1 = (-2, 0)$, $V_2 = (0, 2)$, and $V_3 = (2, -2)$. The problem formulation takes advantage of the fact that all feasible solutions can be expressed as a convex combination of V_1, V_2, and V_3 in the form

$$x = aV_1 + bV_2 + cV_3$$

where a, b, and c are nonnegative coefficients and satisfy the following linear equality constraint:

$$a + b + c = 1$$

Therefore, the solution encoding 'R3' is used with the three elements corresponding to the values of a, b, and c. Note that this strategy can be generalized to any solution domain that can be specified by a convex hull. An additional 'R2' segment is also created to store the corresponding x value. In the program, **function sixhump** computes the objective value. Lower and upper bounds of 0 and 1, respectively, are used to ensure that solution elements are nonnegative. Violations of the linear equality constraint are fixed by a simple repair strategy, implemented in **function sixhump**: the sum of the solution elements is computed, and each element is divided by the sum, so that the sum of the new elements is 1. For the special case of all elements equal to 0, equal values are assigned to the solution elements.

```
proc ga seed = 5555;
call SetEncoding('R3R2');
npoints = 3;
array cvxhull[3,2] /nosym ( -2 0
                             0 2
                             2 -2 );
/* Objective function */
function sixhump(selected[*],cvxhull[*,*],npoints);

  /* Function has global minimum value of -1.0316
   * at x = {-0.0898  0.7126} and
   *    x = { 0.0898 -0.7126}
   */
  array w[1] /nosym;
  call dynamic_array(w,npoints);
  array x[2] /nosym;

  call ReadMember(selected,1,w);

  /* make sure that weights add up to 1 */
  sum = 0;
  do i = 1 to npoints;
    sum + w[i];
  end;

  /* if all weights 0, then reinitialize */
  if sum=0 then do;
    sum = npoints;
    do i = 1 to npoints;
       w[i] = 1;
    end;
  end;

  /* re-normalize weights */
  do i = 1 to npoints;
```

```
        w[i] = w[i] / sum;
      end;

      /* convert weights to x-coordinate form */
      x[1] = 0;
      x[2] = 0;
      do i = 1 to npoints;
        x[1] + w[i] * cvxhull[i,1];
        x[2] + w[i] * cvxhull[i,2];
      end;

      /* write out x coordinates to second segment */
      call WriteMember(selected,2,x);

      /* compute objective value */
      r = (4 - 2.1*x[1]**2 + x[1]**4/3)*x[1]**2 + x[1]*x[2] +
        (-4 + 4*x[2]**2)*x[2]**2;
      return(r);
    endsub;

    call SetObjFunc('sixhump',0);

    array lower[1] /nosym;
    array upper[1] /nosym;

    call dynamic_array(lower, npoints);
    call dynamic_array(upper, npoints);
    do i = 1 to npoints;
      lower[i] = 0;
      upper[i] = 1;
    end;
    call SetBounds(lower, upper, 1);
    call SetMutUniform(1);
    call SetMutProb(0.05);

    call SetCross2Point(0.9);
    call SetCrossProb(1.0);

    call SetSelTournament(2);
    call SetElite(3);

    call Initialize('DEFAULT', 100);
    call ContinueFor(35);
    run;
```

Note that this problem uses the standard genetic operators and default initialization, even though they generate solutions that violate the constraints. This is possible because all solutions are passed into the user objective function for evaluation, where they are repaired to fit the constraints. The output is shown in Output 1.2.1.

Output 1.2.1. Nonlinear Objective with Constraints Using Repair Mechanism

```
                         PROC GA Optimum Values

                                Objective

                              -1.031628453


                               Solution
                   Segment     Element           Value

                         1           1     0.4052493015
                         1           2     0.9213856377
                         1           3     0.3307959879
                         2           1      -0.08984183
                         2           2      0.712656727
```

This objective function has a global minimum at -1.0316, at two different points: $(x_1, x_2) = (-0.0898, 0.7126)$ and $(x_1, x_2) = (0.0898, -0.7126)$. The genetic algorithm can converge to either of these minima, depending on the random number seed set by the SEED= option.

Example 1.3. Quadratic Objective with Linear Constraints, Using Bicriterion Approach

This example (Floudas and Pardalos 1992) illustrates the bicriterion approach to handling constraints. The problem has nine linear constraints and a quadratic objective function.

Minimize

$$f(x) = 5\sum_{i=1}^{4} x_i - 5\sum_{i=1}^{4} x_i^2 - \sum_{i=5}^{13} x_i$$

subject to

$$\begin{aligned}
&2x_1 + 2x_2 + x_{10} + x_{11} \leq 10\\
&2x_1 + 2x_3 + x_{10} + x_{12} \leq 10\\
&2x_1 + 2x_3 + x_{11} + x_{12} \leq 10\\
&-8x_1 + x_{10} \leq 0\\
&-8x_2 + x_{11} \leq 0\\
&-8x_3 + x_{12} \leq 0\\
&-2x_4 - x_5 + x_{10} \leq 0\\
&-2x_6 - x_7 + x_{11} \leq 0\\
&-2x_8 - x_9 + x_{12} \leq 0
\end{aligned}$$

and

$$\begin{array}{ll} 0 \le x_i \le 1, & i = 1, 2, \ldots, 9 \\ 0 \le x_i \le 100, & i = 10, 11, 12 \\ 0 \le x_{13} \le 1 & \end{array}$$

In this example, the linear constraint coefficients are specified in the SAS data set lincon and passed to the GA procedure with the MATRIX1= option. The upper and lower bounds are specified in the bounds data set specified with a DATA1= option, which creates the array variables upper and lower, matching the variables in the data set.

```
/* Input linear constraint matrix */
data lincon;
   input A1-A13 b;
   datalines;
 2  2  0  0  0  0  0  0  0  1  1  0  0 10
 2  0  2  0  0  0  0  0  0  1  0  1  0 10
 2  0  2  0  0  0  0  0  0  0  1  1  0 10
-8  0  0  0  0  0  0  0  0  1  0  0  0  0
 0 -8  0  0  0  0  0  0  0  0  1  0  0  0
 0  0 -8  0  0  0  0  0  0  0  0  1  0  0
 0  0  0 -2 -1  0  0  0  0  1  0  0  0  0
 0  0  0  0  0 -2 -1  0  0  0  1  0  0  0
 0  0  0  0  0  0  0 -2 -1  0  0  1  0  0
;

/* Input lower and upper bounds */
data bounds;
   input lower upper;
   datalines;
0  1
0  1
0  1
0  1
0  1
0  1
0  1
0  1
0  1
0  100
0  100
0  100
0  1
;


proc ga lastgen = out matrix1 = lincon
        seed = 12345 data1 = bounds;
```

Note also that the LASTGEN= option is used to designate a data set to store the final solution generation.

Next the solution encoding is specified, and a user function is defined and designated as the objective function.

```
call SetEncoding('R13R3');

nvar = 13;
ncon = 9;

function quad(selected[*], matrix1[*,*], nvar, ncon);
  array x[1] /nosym;
  array r[3] /nosym;
  array violations[1] /nosym;

  call dynamic_array(x, nvar);
  call dynamic_array(violations, ncon);

  call ReadMember(selected,1,x);

  sum1 = 0;
  do i = 1 to 4;
    sum1 + x[i] - x[i] * x[i];
  end;

  sum2 = 0;
  do i = 5 to 13;
    sum2 + x[i];
  end;

  obj = 5 * sum1 - sum2;

  call EvaluateLC(matrix1,violations,sumvio,selected,1);
  r[1] = obj;
  r[2] = sumvio;
  call WriteMember(selected,2,r);

  return(obj);
endsub;
call SetObjFunc('quad',0);
```

The SetEncoding call specifies two real-valued segments. The first segment, R13, holds the 13 variables, and the second segment, R3, holds the two objective criteria and the marker for Pareto optimality. As described in the “Defining an Objective Function” section on page 55, the first parameter of the objective function is a numeric array that designates which member of the solution population is to be evaluated. When the quad function is called by the GA procedure during the optimization process, the matrix1, nvar, and ncon parameters receive the values of the corresponding global variables; nvar is set to the number of variables and ncon is set to the number of linear constraints. The function computes the original objective as the first objective criterion, and the magnitude of constraint violation as the second. With the first dynamic_array call, it allocates a working array, x, large enough to hold the number of variables, and a second array, violations, large enough to tabulate each constraint violation. The ReadMember call fills x with the elements of the first segment of the solution, then the function computes the original objective $f(x)$. The EvaluateLC call is used to compute the linear constraint violation. The objective and sum of the constraint violations are then stored in the array r, and written back to the second segment of the solution with the WriteMember call. Note that the third element of r is not modified, because that element of the segment is used to store the Pareto-optimality mark, which cannot be determined until all the solutions have been evaluated.

Next, a user routine is defined and designated to be an update routine. This routine is called once at each iteration, after all the solutions have been evaluated with the quad function.

```
subroutine update(popsize);
  /* find pareto-optimal set */

  array minmax[3] /nosym (-1 -1 0);
  array results[1,1] /nosym;
  array scratch[1] /nosym;

  call dynamic_array(scratch, popsize);
  call dynamic_array(results,popsize,3);

  /* read original and constraint objectives, stored in
   * solution segment 2, into array */
  call GetSolutions(results,popsize,2);
  /* mark the pareto-optimal set */
  call MarkPareto(scratch, npareto,results, minmax);

  /* transfer the results to the solution segment */
  do i = 1 to popsize;
    results[i,3] = scratch[i];
  end;

  /* write updated segment 2 back into solution population
  */
  call UpdateSolutions(results,popsize,2);

  /* Set Elite parameter to preserve the first 15 pareto-optimal
   * solutions
  */
  if npareto < 16 then
     call SetElite(npareto);
  else
     call SetElite(15);
endsub;

call SetUpdateRoutine('update');
```

This subroutine has one parameter, popsize, defined within the GA procedure, which is expected to be the population size. The working arrays results, scratch, and minmax are declared. The minmax array is to be passed to a MarkPareto call, and is initialized to specify that the first two elements (the original objective and constraint violation) are to be minimized, and the third element is not to be considered. The results and scratch arrays are then dynamically allocated to the dimensions required by the population size.

Next, the results array is filled with the second segment of the solution population, with the GetSolutions call. The minmax and results arrays are passed as inputs to the MarkPareto call, which returns the number of Pareto-optimal solutions in the npareto variable. The MarkPareto call also sets the elements of the scratch array to 1 if the corresponding solution is Pareto-optimal, and to 0 otherwise. The next loop then records the results in the scratch array in the third column of the results array, effectively marking the Pareto-optimal solutions. The updated solution segments are written back to the population with the UpdateSolutions call.

The final step in the update routine is to set the elite selection parameter to guarantee the survival of at least a minimum of 15 of the fittest (Pareto-optimal) solutions through the selection process.

With the following statements, a routine is defined and designated as a fitness comparison routine with a SetSel call. This routine works in combination with the update routine to evolve the solution population toward Pareto optimality and constraint satisfaction.

```
function paretocomp(selected[*]);
  array member1[3] /nosym;
  array member2[3] /nosym;

  call ReadCompare(selected,2,1, member1);
  call ReadCompare(selected,2,2, member2);

  /* if one member is in the pareto-optimal set
   * and the other is not, then it is the
   * most fit
  */
  if(member1[3] > member2[3]) then
     return(1);
  if(member2[3] > member1[3]) then
     return(-1);

  /* if both are in the pareto-optimal set, then
   * the one with the lowest constraint violation
   * is the most fit
  */
  if(member1[3] = 1) then do;
     if member1[2] <= member2[2] then
        return(1);
     return( -1);
  end;

  /* if neither is in the pareto-optimal set, then
   * take the one that dominates the other
  */
```

```
      if (member1[2] <= member2[2]) &
         (member1[1] <= member2[1]) then
         return(1);
      if (member2[2] <= member1[2]) &
         (member2[1] <= member1[1]) then
         return(-1);

      /* if neither dominates, then consider fitness to be
       * the same
      */
      return( 0);
   endsub;

   call SetSelTournament(2,'paretocomp');
```

The **paretocomp** subroutine is called in the selection process to compare the fitness of two competing solutions. The first parameter, selected, designates the two solutions to be compared.

The ReadCompare calls retrieve the second segments of the two solutions, where the objective criteria are stored, and writes the segments into the member1 and member2 arrays. The logic that follows first checks for the case where only one solution is Pareto-optimal, and returns it. If both the solutions are Pareto-optimal, then the one with the smallest constraint violation is chosen. If neither is Pareto-optimal, then the dominant solution is chosen, if one exists. If neither solution is dominant, then no preference is indicated. After the function is defined, it is designated as a fitness comparison routine with the SetSel call, which also specifies tournament selection with a tournament size of 2.

Next, subroutines are defined and designated as user crossover and mutation operators.

```
   /* set up crossover parameters */

   subroutine Cross1(selected[*], alpha);
      call Cross2Point(selected,1, alpha);
    endsub;
   call SetCrossRoutine('Cross1',2,2);
   alpha = 0.5;
   call SetCrossProb(0.8);

   /* set up mutation parameters */

   subroutine Mut1(selected[*], delta[*]);
     call MutDelta(selected,1,delta,1);
   endsub;
   call SetMutRoutine('Mut1');
   array delta[13] /nosym (.5 .5 .5 .5 .5 .5 .5 .5 .5 10 10 10 .1);
   call SetMutProb(0.05);
```

These routines execute standard genetic operators *2point* for crossover and *delta* for mutation; see the "Using Standard Genetic Operators" section on page 44 for a description of each. The alpha and delta variables defined in the procedure are passed as parameters to the user operators, and the crossover and mutation probabilities are set with the SetCrossProb and SetMutProb calls.

At this point, the GA procedure is directed to initialize the first population and begin the optimization process.

```
/* Initialize first population */
call SetBounds(lower, upper);
popsize = 100;
call Initialize('DEFAULT',popsize);

call ContinueFor(500);
run;
```

First, the upper and lower bounds are established with values in the lower and upper array variables, which were set up by the DATA1= option in the PROC GA statement. The SetBounds call sets the bounds for the first segment, which is the default if none is specified in the call. The desired population size of 100 is stored in the popsize variable, so it will be passed to the update subroutine as the popsize parameter. The Initialize call specifies the default initialization, which generates values randomly distributed between the lower and upper bounds for the first encoding segment. Since no bounds were specified for the second segment, it is filled with zeros. The ContinueFor call sets the requested number of iterations to 500, and the RUN statement ends the GA procedure input and begins the optimization process. The output of the procedure is shown in Output 1.3.1.

Output 1.3.1. Bicriteria Constraint Handling Example Output

```
                   Bicriteria Constraint Handling Example

                          PROC GA Optimum Values

                                Objective

                              -14.99885993


                              Solution
                    Segment    Element          Value

                          1          1              1
                          1          2              1
                          1          3              1
                          1          4              1
                          1          5              1
                          1          6   0.9999999964
                          1          7   0.9999999915
                          1          8   0.9999987482
                          1          9   0.9999999877
                          1         10   2.9999999783
                          1         11   2.9997926643
                          1         12   2.9990685636
                          1         13              1
                          2          1   -14.99885993
                          2          2              0
                          2          3              1
```

The minimum value of $f(x)$ is -15 at $x^* = (1,1,1,1,1,1,1,1,1,3,3,3,1)$.

References

Floudas, C. A. and Pardalos, P. M. (1992), *Recent Advances in Global Optimization*, Princeton, NJ: Princeton University Press.

Michalewicz, Z. (1996), *Genetic Algorithms + Data Structures = Evolution Programs*, New York: Springer-Verlag.