# GenET

## USER's MANUAL
## for version 1.0



Cezary Z. Janikow

Department of Mathematics and Computer Science
University of Missouri – St. Louis
St. Louis, MO 63121
janikow@radom.umsl.edu

Cezary Z. Janikow, September 20, 1993

# GenET USER's MANUAL: for version 1.0

## 1 PREFACE

**GenET** (**Gen**eric **E**volutionary **T**oolbox) is a generic evolutionary algorithm, a toolbox for fast development of genetic algorithm applications (GAs in the very broad sense) and for research in evaluating different evolutionary models, operators, *etc*. This manual describes version **GenET.1.0**, which is an initial release in *beta* stage.

The general theme of this tool is:

> In developing a GA application, the user should only concentrate on proposing the best representation for the problem – which defines the search space – and the operators best utilizing specific properties of the problem in the so-defined space.

This version is being distributed to all interested parties for evaluation (and applications). Please report to `janikowGA@radom.umsl.edu` with `Subject: GenET`

- all bugs, with Body: `GenET BUGS`
- suggestions for modifications, with Body: `GenET MODIFICATIONS`
- all suggestions for additional models/options to be implemented in future releases, with Body: `GenET EXTENSIONS`
- send your operators (with chromosome definitions if new or name of existing library that the operators apply to), with Body: `GenET OPERATORS`
- any other ideas/comments, with Body: `GenET COMMENTS`

Please try to spell the messages correctly for automatic handling. This manual is also available in hard copy:

- to request one, send your address, with Body: `GenET MANUAL HARDCOPY`
- to request email with postscript, send a message with Body: `GenET MANUAL POSTSCRIPT`

Please acknowledge this manual as a technical report of our department in any publication referring to **GenET**. Please, send electronic versions of such publications with Body: `GenET PUBLICATIONS` or hard copies to the physical address. If you find it useful and would like to see the algorithm completed and extended, please send a note with Body: `GenET COMMENTS` (this is important to ensure some support for the development and maintenance).

## 2 GENERAL OBJECTIVES

The general objectives of releasing this ongoing project into public domain are to:

1. Provide a tool for fast development of evolutionary applications to advance such applications. This generic package differs from others widely available in its current and anticipated capabilities, and in its complete separation from problem-specific chromosome representation and operators – with a standard interface.
2. Provide a research tool for testing and comparing different evolutionary models.
3. Provide a research tool for evaluating different proposed operators. There has been a long discussion, and long list of publications, about advantages of different GA operators. **GenET** provides a tool for studying (or applying) them in various applications by providing library capabilities for representations and operators, and by providing capabilities to adapt operators based on their utilities.

4. Establish such libraries of representations with operators, available with future releases, to provide an even higher level abstraction in evolutionary applications (current version provides FP representation with sample operators).
5. Provide a toolbox for individual extensions.

To facilitate research capabilities, GenET provides a choice of dumping run traces to spreadsheet files (see Section 4.7)

### 2.1 GenET as Tool for Fast Development of Applications

Most people ever attempting any GA application should agree that major time-consuming obstacles include: lack of good software engineering (design, programming, maintenance) skills of the developer or his team, implementing and testing the desired GA model, deciding on the best representation, and implementing operators for such representation. Unfortunately, later changes to representation often require modifications of other parts of the algorithm. And vice-versa, later changes of the model often require changes in the operators and the representation. Such changes take often place since it is difficult to assess the best model for a given problem a priori. Moreover, after a successful deployment, another time-consuming process involves assessing the operators and selecting the best probabilities. This is extremely difficult as simulations take time, there is often a strong interrelationship among different operators, and some operators are better or worse at different times during a simulation.

Even assuming good programming skills (which does not necessarily implies good SE skills) still leaves us facing the remaining time-consuming obstacles. Fortunately, the GA model does not have to be aware of the used representation, operators, and evaluation. **GenET** takes this fact to the advantage and it provides a completely problem-independent GA. However, it makes no assumptions on the representation used for an application as most other *generic* algorithms do. To develop a new application, the user has to only define the representation, all desired operators, and the evaluation function. Period. The shifted focus to such problem-specific characteristics allow taking them to full advantage.

**GenET** speeds up GA application developments by up to 99% for those weak software engineers without extensive GA libraries, by about 95% for good developers without such GA libraries, and still by about 50% for these good developers with complete extensive GA libraries (no need to combine them into the desired model). The saving is even better when other models/options are desired later on.

### 2.2 GenET as Research Tool for Working with Different GA Models

It is extremely difficult to decide a priori the best model for a given problem. **GenET** reduces any complexity related to implementation of different models by providing many models by simple selection. This capability will allow existing models, and new ones as they will be implemented in future versions, to be experimentally compared on a variety of classes of problems. And, of course, this approach allows for instantaneous testing of different models for a given application at hand.

ES have been developed and tuned for FP parameter optimization, while GAs were designed to be problem-independent search methods. Both approaches offered certain advantages. Independently from each other, these two general methods started to diverge from the originals as new extensions were being proposed. Interestingly, they often started borrowing ideas from the other methods. **GenET** provides capabilities to study a very broad class of such new models.

### 2.3 GenET as Research Tool for Evaluating Operators and Representations

There is an ongoing debate about the utility of different representations for classes of problems, and the utility of different operators given a specific representation. **GenET**'s fast development cycle will allow for testing different representations on a wide variety of problems. **GenET**'s adaptive operator probabilities feature allows for direct comparison of qualities of such operators and their utilities on different problems. In additions, this feature speeds up actual applications by cutting the time needed for testing with different operator probabilities.

### 2.4 GenET as Ongoing Project to Further Abstraction of GA Application Developments

Because the main responsibility of a developer using **GenET** is to provide the representation and operators for the representation (in addition to evaluation which is completely problem-specific), we hope to create a library of representations and operators for such representations. Given such library is available with future versions of **GenET**, the application development task may in many instances be further reduced to simple selection of the representation (with its ready display functions) and some of its the operators, and to implementation of the evaluation function only.

### 2.5 GenET as Toolbox for Individual Extensions

**GenET** has been implemented with functional design – modules have standard interfaces that can be connected or modified to create new models. Header files describe the interfaces. Using these interfaces, the user can create completely new models to test new ideas.

## 3 ENVIRONMENT REQUIREMENTS

Any **ANSI C** compiler that allows for full function prototypes should compile **GenET**. 16-bit computers without standard `RAND_MAX` definition will have to modify this constant in `main.h` file.

The release was tested using `gcc` on a Data General and using ATT's `CC` (object-oriented compiler is not needed as **GenET**'s design is functional) on a `SparcII`, in both cases with `UNIX 4.1.x`. In general, the project should compile on any ANSI C compiler. Some environments provide different `make` capabilities – these files may have to modified individually for such environments.

## 4 CURRENT CAPABILITIES

### 4.1 Choice of Generation-to-Generation Model

1. **Generational model**, where selection alone (with replacement) generates new intermediate population and only then operators alter the intermediate population. In the generational model, an offspring may be modified by a number of different operators, and the selection with replacement itself can be seen as a meta-operator.
2. **Operational model**, where operators act on members of the population to produce (tentative) members of the new population. In this model, each new offspring is a product of exactly one operator. The offspring may compete with each other, or even with the parents, for survival into the next population. As such, this model is characterized, in addition to offspring being products of exactly one operator, by two selections: one to select, with replacement, parents to generate the intermediate offspring population, and another to select, without replacement, the next population. The next population can be selected from only the intermediate offspring population (as in (n,m≥n)-ES) or from a union of the parent and the offspring population ((n,m+n)-ES).

Both models, given the other choices, allow a wide range of cases. For example, you may generate only one offspring per generation, place it over the weakest chromosome by using (n, n+1)-model with "razor cut" for the survival selection. Alternatively, you may replace a random chromosome, or use a roulette to decide that.

### 4.2 Choice of Parent Selection

In any model, parents can be selected

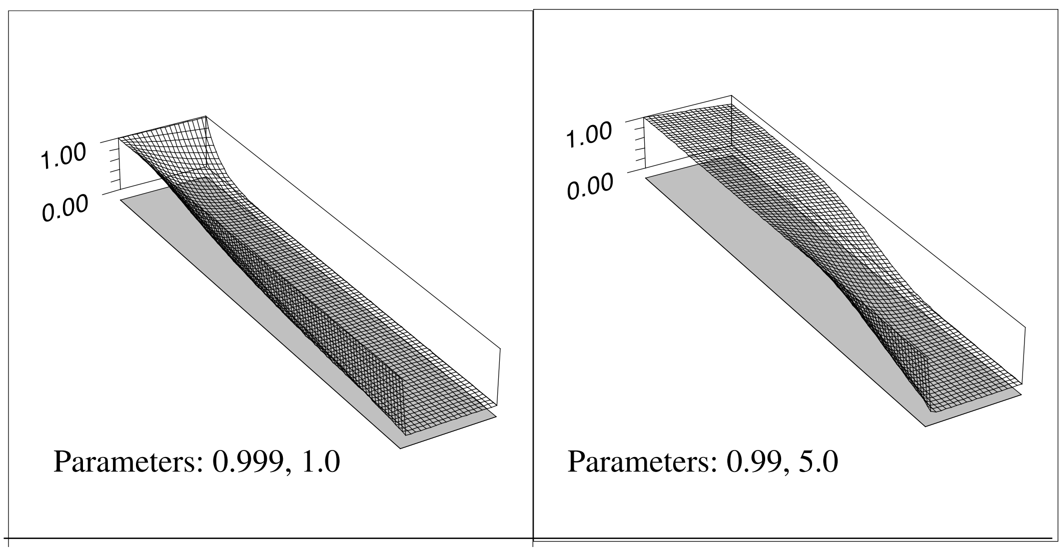
1. Randomly (the random selection is probably not very useful for the generational model where this is the only selection used – in the operational model, there are two selections per generation and one of them may be random).
2. By roulette.

### 4.3 Choice of Survival Selection (operational model)

In the operational model, a second selection is needed to select the population surviving into the next generation. You have a choice of:

1. Random selection (not a very good choice if the parent selection is random as well).
2. By roulette.
3. By a razor cut. This is the most often used method for the operational models – the weakest members are removed.

**Figure 1** Simulated rank-based pressures for two different runs. The first run starts with almost uniform selective pressure, which however increases much more rapidly.




### 4.4 Choice of Selective Pressure

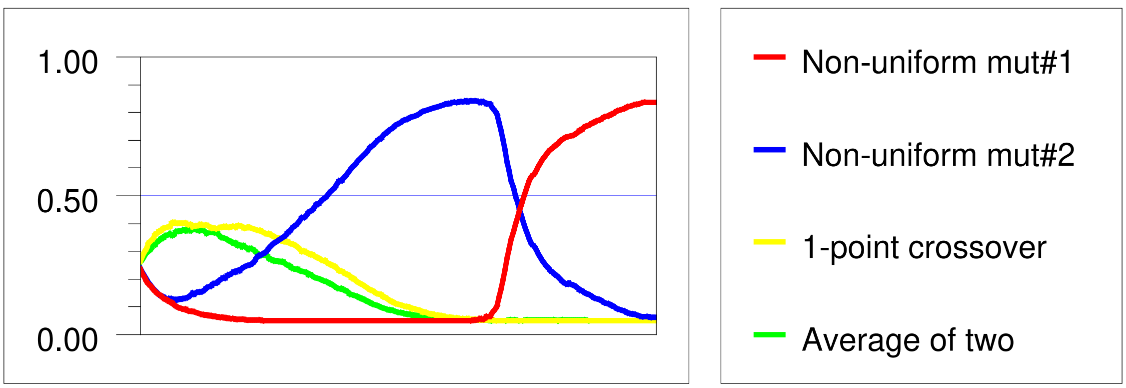
For roulettes, the selective pressure can be controlled by

1. Pure evaluations. Negative evaluations are dealt with by shifting all the evaluations up the scale.
2. Ranking. The implemented ranking schema is dynamic and controlled by two parameters, which allow setting the pressure schedule as desired. See Figure 1 for examples.

### 4.5 Choice of Static or Adaptive Operator Probabilities

**GenET.1.0** can adapt operators. This adaptation, as implemented, is based on the operators' utilities to produce offspring capable of survival in the population. Since this is a generic tool, only operator probabilities (the only universal feature of an operator) can be adapted (other specific features can be adapted as well, but such mechanisms would have to be implemented with the operators). See Figure 2 for an example of self-adaptation during optimization of a dynamic control problem in FP representation using two different mutations and crossovers. For this run, all four operators were given equal initial probability in the operational model. It can be seen that initially both crossovers did most of global exploration, then one of the mutations (that with wider standard deviation) performed best, and the result was fine-tuned by the finer mutation. On this run, the minimal probability was set to 0.05 to avoid operators with low initial utility (as the first mutation) from being completely turned off.

**Figure 2** Adaptive operators probabilities of four operators during a sample run of one of the sample problems. See how the operators are used only when their utility is high.




### 4.6 Stopping Conditions

Currently, you may set up the runs for a number of generations or a number of seconds.

### 4.7 Tracing Runs

**GenET.1.0** allows you to generate a number of trace files suitable for visualization:

1. The rank-based selective pressure scheduled for the run.
2. Probabilities of operators when adaptation is used.
3. Intermediate best chromosomes.
4. Intermediate best evaluations.

### 4.8 Operators

In the generational model, an offspring can be a result of application of a number of operators. Moreover, some operators were traditionally designed so that they could apply multiple number of times to the same chromosome – e.g., mutation with its probability defined over an individual gene and not a chromosome. In the operational model, however, each offspring is a result of exactly one operator application, and all the probabilities refer to whole chromosomes. To be able to use the same library operators and yet provide a choice of a model, we use the operator probability as the probability of application to a chromosome. In the generational model, different operators can still apply to the same chromosome. As to multiple applications of a single operator to a chromosome, it is left up to the operator to accomplish that. Note that this scenario allows for such multiple applications in both the models. More details about this model will be provided when its implementation is completed and integrated with **GenET**.

# 5 HOW TO OBTAIN THE PACKAGE

1. `% ftp radom.umsl.edu`, using anonymous `ftp`. Change directory to `/var/ftp`. There are two binary files: `GenET.tar.Z` containing the software and `userMan.ps.Z` containing the user manual, both in compressed binary formats. Note that `GenET.tar.Z` also contains the user manual so that only one file needs to be acquired.
2. If you obtained the manual only, `% uncompress userMan.ps.Z` and then `% lpr userMan.ps`. Make sure you have no node called `userMan.ps`.
3. If you obtained `GenET.tar.Z`, put it into your file system in the parent of the anticipated directory. Make sure there is no node called `GenET`.
4. `% uncompress GenET.tar.Z`, then `> tar xfv GenET.tar`. This will create **GenET** subdirectory.

The user's manual alone can also be request by email – see section 1.

# 6 MODULE INTERFACES FOR APPLICATION DEVELOPMENT

There is a standard interface between the algorithm and the problem/representation/operators needed to implement an application. Standard interfaces are fixed for all problems regardless of their representation and configuration. If any specific representation or operators require any additional interface, this is be referred to as the non-standard interface.

## 6.1 Standard Interfaces

### 6.1.1 Standard interface between algorithm and problem

The standard interface between GenET algorithm and a problem is as defined in `ALG/problem.h`:. The user developing a new application is responsible for providing the implementations in the problem file.

```
/* evalChrom(chrom) returns fitness of the chromosome */
extern double evalChrom(void *chrom);

/* initChrom(chrom) is problem-specific but only for the given representation */
/* fp is the input file pointer */
extern void initChrom(genericChromType *chrom, FILE *fp);
```

The `genericChromType` is defined in `ALG/pop.h` as follows:

```
typedef struct { double eval;
   void *chrom;
   double *operProbs;
   double *operApplCounter;
   double operApplCounterTotal;
   void *roulette;
} genericChromType;
```

### 6.1.2 Standard interface between algorithm and representation

The standard interface between **GenET** and representation is as defined in `ALG/representation.h`. While using a representation from the existing library, the implementations are provided. For example, the provided FP representation is implemented in `doubleVector/REP/representation.c`. While developing an application with a new representation, the user has to provide the implementations.

```
/* setRepSpecifics() is used to set up any other information specific */
/* to a representation */
extern void setRepSpecifics(FILE *input);

/* makeChrom() allocates storage for a chrom and initializes it; fp is input */
extern genericChromType *makeChrom(FILE *input);

/* dspChrom() is provided if desired to display chromosomes */
extern void dspChrom(FILE *output, void *chrom);
```

In addition, the actual definition of the representation is to be provided through header file in `REP/representation.def`. This file might be empty if the representation does not require any explicit representation definition, as in the case with the FP representation which implicitly assumes that chromosomes are arrays of doubles, with number of genes provided on the non-standard interface. Alternatively, a chromosome could have been defined as

```
typedef struct {
   int numGenes;
   double *geneVector;
} chromType;
```

Then, an explicit representation would have been defined as a type. While this approach is generally preferable, many conversions can be avoided by following the existing implementation with `numGenes` available as non-redundant global in the problem file.

### 6.1.3 Standard interface between algorithm and operators

The standard interface between **GenET** and operators is as defined in `ALG/operators.h`. While using a representation from the existing library, the implementations are provided. For example, few operators for FP representation are implemented in `doubleVector/REP/operators.c` While developing an application with a new representation, the user has to provide the implementations (as well as the actual operators).

```
/* operators.h defines standard interface to current rep's operators */
/* getOperName(operNum) returns name of operator operNum (of size NAMESIZ) */
extern char *getOperName(int operNum);

/* getOperAddr(operNum) returns address of operator operNum for indirect */
/* invocationrs */
extern operAddrType getOperAddr(int operNum);

/* getNumOpers() returns the number of implemented operators */
extern int getNumOpers(void);

/* setOperSpecifics(fp,operNum) sets any operator-specific information */
/* for operNum. Specific information includes any parameters other than */
```

```
/* probability of application to a chromosome */
extern void setOperSpecifics(FILE *fp, int operNum);
```

### 6.2 Non-Standard Interface

Depending on some properties of the representation and the problem at hand, some non-standard interface might be needed in addition to the standard interface. This interface will provide communication for issues specific to a given representation. For example, the FP representation, implemented in *representationDirectory*=doubleVector, needs to provide the number of genes in a chromosome and gene domains:

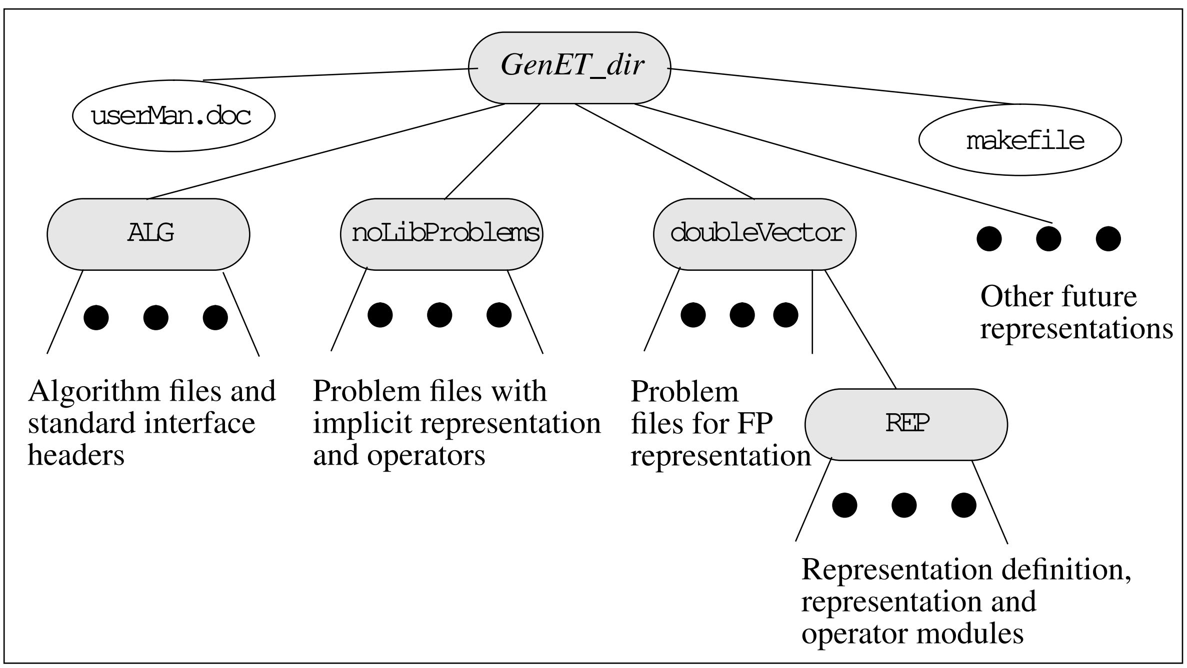
```
/* representationNS.h contains problem-specific, non-standard interface to */
/* representation; here, domains are provided as doubles via LowerDom[] */
/* and UpperDom[]; vector length in provides via NumGenes */

extern double *LowerDom;
extern double *UpperDom;
extern int NumGenes;
```

The actual definitions should be provided in the representation file. For example, doubleVector/representation.c implements storage for both the domains (to be read-in) and NumGenes.

## 7 CREATING APPLICATIONS

**Figure 3 GenET**'s hierarchical organization.




**GenET** is organized to provide for incremental library growth and ease of management. Its structure is shown in Figure 3. The main directory *GenET_dir* (which would be called GenET after the steps of section 5) contains user's manual and the main makefile. The ALG directory contains all sources for the generic algorithm and standard interface headers. noLibProblems is the suggested directory to place your problems with implicitly implemented representation and operators. The remaining directories are libraries of representations and operators. For each representation, its directory will contain problem files using the representation and a subdirectory `REP` containing definitions and implementations of the representation and its operators.

Following the organization, a new application can be created in one of two ways: implicit or explicit, depending whether the desired representation with operators is present in the library.

### 7.1 Creating New Explicit Library Representation

If library does not contain the desired representation and the representation is general enough to justify explicit creation, create a new representation directory *newRep* as one of the future representations in Figure 3, then implement the representation there. All problem files using this new representation are to be placed in *newRep* directory.

A representation is defined and implemented in *newRep*/`REP` directory using the following files: `representation.def`, `representation.c`, `operators.c`. Any nonstandard interface is shared by `representationNS.h` or `operatorsNS.h`.

#### 7.1.1 `representation.def`

This file defines the representation if an explicit type definition for the chromosome is needed. For example, doubleVector uses implicit type for the chromosome so that the file is empty, but the same representation could have been defined with an explicit definition (see section 6.1.2).

#### 7.1.2 `representation.c`

This file implements the standard interface between the algorithm and the representation, as defined in section 6.1.2: `setUpRepresentation()`, `makeChrom()`, and `dspChrom()`.

`setRepSpecifics(FILE*)` is used to set up any values specific to this particular representation. The file pointer is the current input stream, either the standard or a file. Use it to set any values needed of the particular representation. Such values must then be made available to other modules that would need them through a non-standard interface `representationNS.h`. For example, `doubleVector` requires the number of genes and domains for each gene.

`makeChrom(FILE*)` is used to allocate and initialize storage for whatever the chromosome is in the representation. For initializations to be problem-specific, this function can call `initChrom(FILE*)`, available on the standard interface to a problem. The file pointer is again the current input stream.

`dspChrom(FILE*, void*)` is used to display a chromosome so that the algorithm can utilize them to provide such displays on the user interface. If no displays are needed in the representation for whatever reason, this might be an empty function. The file pointer is the current output stream, and the void pointer is the chromosome to be displayed – the **GenET** algorithm does not know how to interpret a chromosome.

#### 7.1.3 `operators.c`

This is the most complex file to be implemented since in addition to the standard interface implementation it must also provide the operators (operators are not available on the standard interface, they are available implicitly by addresses).

We will first discuss implementation of the standard interface (see section 6.1.3).

`getOperName(int)` is used to retrieve the name of the operator whose number is supplied. These names are used to provide a more meaningful user interface. The implementation can be explicit with a `switch` statement, or implicit by referring to the list of such names (as done in doubleVector representation).

`getOperAddr(int)` is used to retrieve address of the operator whose number is specified and provides for the implicit usage. The implementation can be explicit with a `switch` statement, or implicit by referring to the list of such names (as done in doubleVector representation). Make sure that addresses returned by this function correspond to names returned by the previous function.

`getNumOpers(void)` is used to retrieve the number of operators implemented in this library so that the user interface that is used to interactively select those desired for a particular run know the total number.

`setOperSpecifics(FILE*,int)` is used similarly to `setRepSpecifics()` in section 7.1.2. This capability allows to implement operators that have parameters (other than the general probability of application to a chromosome) that are specifics to operators are interactively set for each run. The file pointer provides the current input stream, and the second operator is the operator number in question. Any values set up here should be global restricted (`static`) in this file. Because they are not needed anywhere else, no `operatorsNS.h` non-standard definition is needed as `representationNS.h` was. The implementation is irrelevant – again, a `switch` statement or indirect invocation by an address (as done in the `doubleVector` representation) can be used.

Now we proceed to discussing implementation of the operators. Operators are not provided explicitly on the standard interface between the algorithm and the operators – this interface is accomplished by indirect invocations using addresses supplied with `getOperAddr(int)`. Accordingly, all operators are defined with restricted linkage (`static`). The general form is:

`static int` *operName* `(genericChromType *parentChrom, genericChromsType newChroms, int maxNumOff, int curNumOff)`, where

- `parentChrom` is a pointer to the parent chromosome for the operator. `genericChromType` was defined in section 6.1.1. Only this one parent is provided. If more parents are needed, the standard interface provides `getParent()` available as `genericChromType *getParent(void)`, which returns a parent according to the currently used parent selection method.
- `newChroms` is the being created new population – its type is defined as `typedef genericChromType **genericChromsType` (in other words, it is a pointer to an array of generic chromosomes).
- `maxNumOff` is the size of the new population storage.
- `curNumOff` is the current number of offspring the new population storage. Any operator can produce at most `maxNumOff-curNumOff` offspring in one firing.

Each operator should be aware of its number (those used to identify operators in `getOperName()` and `getOperAddr()` – make sure they are consistent). The implementation should realize the action of the operator. If more than one parent is needed, they are obtained using `getParent()`. The offspring is (are) placed in `newChroms`, starting at location `curNumOff`. The new population storage is allocated for all the chromosomes, so only the proper values must be set. Following each operator action and just before returning from the operator, a finishing function must be called, once for each generated offspring. The form of this function is:

`void finishOper(genericChromType *newChrom, int operNum, int *curNumOff, int numParents, ...)`, where

- `newChrom` is the generated offspring

- `operNum` is the number of the current operator
- `curNumOff` is the same as the current parameter. Note that `curNumOff` is not supposed to be incremented explicitly in the operator – each call to `finishOper()` does a single increment.
- `numParents` is the number of parents used to generate the offspring
- ... is a variable length list of parents. It would always include `parentChrom`, and also any other parents obtained to generate the offspring.

Finally, each operator returns the number of offspring it placed in the new population.

After the explicit representation is created, creation of an application follows the description of section 7.2.

### 7.2 Creating Applications with Existing Explicit Library Representation

Explicit representations are provided in the library (currently only FP representation with four operators is available in `doubleVector`). When creating a new application for an existing representation available in *repDirectory*, it is necessary to create a problem source file implementing `evalChrom()` and `initChrom()`, that is the standard algorithm-problem interface (see section 6.1).

`double evalChrom(void*)` takes a void pointer to a chromosome and returns its evaluation as double. For example, if the objective is to maximize/minimize a sum of real parameters using `doubleVector` library, the function might be:

```
double evalChrom(void *c)
{   double *chrom=(double*)c;
    double sum=0;
    register int i;
    for (i=0; i<NumGenes; i++)
        sum+=chrom[i];
    return(sum);
}
```

Note that `NumGenes` is provided by the non-standard interface in that representation library (along with domains `LowerDom` and `UpperDom`, in `doubleVector/representationNS.h`).

`void initChrom(genericChromType *, FILE *)` takes a generic chromosome pointer (see section 6.1) and initializes its values. For example, the dynamic control problem in `doubleVector/dynCntrl.c` uses a random initialization in the domain:

```
void initChrom(genericChromType *c, FILE *fp)
{   register int i;
    double *chrom=(double*)c;
    for (i=0; i<NumGenes; i++)
        chrom[i]=LowerDom[i]+probRand()*(UpperDom[i]-LowerDom[i]);
}
```

The file pointer provides the current input stream if the initialization is to be done from a file. This problem file is to be placed in the representation directory, *repDirectory/problemName*`.c`, along with other problem files for this particular representation. To compile the problem file with the representation and the algorithm, execute the command from the *GenET_dir*:

```
% make library [CC=yourCompiler] [LD=yourLinker] REPDIR=repDirectory [PROBLEM-
```

NAME=*problemName*] [EXENAME=*executable*]
with defaults as follows:

- CC=your system's default compiler
- LD=your system's default linker
- PROBLEMNAME=problem in REPDIR (.c extension is implicit)
- EXENAME=PROBLEMNAME.run (in REPDIR)

For example, to compile the dynamic control problem with the doubleVector representation, do:

```
% make library REPDIR=doubleVector PROBLEMNAME=dynCntrl
```

which will create *executable*=doubleVector/dynCntrl.run. To execute the *executable*.

a. % *executable* – start an interactive run.
b. % *executable paramFile* – start a non-interactive run with parameters read from *paramFile*. This file must contain a sequence of parameter values, one per line, exactly matching those desired for the run. This option is valuable if it is desired to repeat a certain run – the first run can be interactive with the interactively provided parameters echoed into the *paramFile* file.

### 7.3 Creating Applications with Implicit Representation

When the needed representation and operators are not available in the library and it is not likely that future problems may be using exactly the same representation, creating a new application with implicit representation (out of the library) is the fastest option since you will have to deal with a single file only. Such problem files must contain implementations of all the functions needed on the standard interface (see section 6.1). Suggested directory for such files is noLibProblems, but the file can be placed anywhere. Implementation is analogous to creation of new library and implementing the problem file, except that all functions go into a single file and, therefore, only the standard interface and the operators need to implemented (no no-standard interface). For some more information, follow the example of noLibProblems/simpleExample.c, which also illustrates how to deal with heterogeneous genes in a chromosome.

To compile the problem file, execute the command from the *GenET_dir*:

% make noLibrary [CC=*yourCompiler*] [LD=*yourLinker*] [PROBLEMDIR=*problemDirectory*] [PROBLEMNAME=*problemName*] [EXENAME=*executable*]

- CC=your system's default compiler
- LD=your system's default linker
- PROBLEMDIR=noLibProblems
- PROBLEMNAME=problem in PROBLEMDIR (.c extension is not explicit)
- EXENAME=PROBLEMNAME.run (in PROBLEMDIR)

5. Execute the *executable* as in 7.2.

## 8 USER INTERFACE

There is a number of parameters the user has to specify to run **GenET**. Currently, all needed can be entered interactively or non-interactively from a *paramFile*. Future versions will provide more user-friendly windowing interfaces.

Based on some options, not all parameters will be requested. Below we explain all possible parameters and options needed in the standard algorithm. Specific representations and problems may provide additional user interface to set some problem/representation-specific parameters. NOTE: convention used assumes that *no* answers are given by *0*, *yes* answers by *1*.

- `Do you want to randomize random calls? [0..1]:`
  Answer 1 to randomize the run.
- `Do you want adaptable operator probabilities? [0..1]:`
  Answer 1 to provide operator adaptation during the simulation. This will also serve to evaluate your operators.
- `Give change percent (smaller for smaller changes) [0.010000..1.000000]:`
  Try about 0.1..0.2. Higher number makes the probabilities more adaptive, which may be good only if the time-frame for modifications is not too short. Notice that the proper fraction is related to the length of the operator modification time-frame and its relation to overall simulation length.
- `Do you want to reset oper. counters after modification? (1 is more dynamic) [0..1]:`
  1 is more dynamic as it looks at operator's utility at each operator modification time-frame; 0 causes the utility to be accumulated from the beginning of the run.
- Now, you will be interactively asked for operators to be used from among those available for the problem or the representation.
  `Do you want to use non-uniform mutation, more dense? [0..1]:`
  Answer 1 to use this operator in your run. You must use at least one operator for the operational model to run. For each 1 answer, you will be prompted for the operator's weight:
  `Give initial weight for operator random mutation [0.000000..1.000000]:`
  The entered numbers are weights in relation to one another. That is, two equal weights will yield two same probabilities. In the generational model, these will be the actual application probabilities.
- `Give min operator probability [0.000000..0.240000]:`
  The minimal probability for operators selected for use. This is used when operators are being adapted to prevent some operators from being disabled before their utility increases. The upper bound is 1/number of operators. A good choice is about 20% of the maximal value.
- `Give run time kind 0:seconds 1:iterations [0..1]:`
  Enter 0 if you want the limit to be in clock time (e.g., time is critical in actual application), 1 in number of iterations (more direct comparison of results across different platforms).
- `Give me run time as number of iterations [0..4294967295]:`
  This prompt will either asks for seconds or iterations based on your previous entry.
- `Do you want to set interval for best chrom/eval disp/dump? [0..1]`
  Enter yes if you want evaluations or chromosomes to be periodically displayed or dumped to a disk file. You will be prompted for each specific action and the file name if applicable.
- `Give operator prob modification interval kind 0:seconds 1:iterations [0..1]:`
  How often you want to modify operators based on observed utility. Try the same kind of run as in run kind.
- `Give operator prob modification interval as number of iterations [1..4294967295]:`
  Assuming you selected iteration-based run. This is the operator modification time-frame. Enter shorter time if the change percent was low. A good choice is so that there at least 100

modifications during the run (run time/100).

- `Do you want to display/dump modified operator probs? [0..1]:`
  Enter yes to display or dump to a file the values as they are being modified. You will be prompted for your choice and the file name if applicable.
- `Do you want to minimize? [0..1]?`
  Enter yes for minimizing the objective evaluations.
- `Do you want to display best eval on each improvement? [0..1]:`
- `Do you want 0:Operational 1:Generational model? [0..1]:`
  In operational model. an offspring is a product of exactly one operator; the offspring compete for survival (possibly against parents as well). In generational model, first new population of parents is selected and then operators act on its members.
- `Give population size [1..250]:`
  The population that survives in the operational model, or the general population in the generational model.
- `Give me parent selection kind 0:randomly 1:roulette [0..1]:`
- `Do you want to use both parents and offspring to select end population? [0..1]:`
  In the operational model only – whether the offspring compete against each other only or against the parents as well for survival into the next population.
- `Give new population size [1..10000]:`
  Only in the operational model – the number of offspring generated. If parents do not compete against these offspring for survival, this size must be at least the population size.
- `Give me end pop selection kind 0:randomly 1:roulette 2:razor [0..2]:`
  In the operational model only – how the final survival is conducted. Random or razor are usually used in steady-state models.
- `Do you want to use rank for probabilities? [0..1]:`
  Enter 0 to use direct evaluations to control selection, 1 for ranks. Rank-based selection is controlled by two interactive parameters.
- `Give initial weight for rank-based probs [0.001000..1.000000]:`
  Comes up when ranking is selected - the first parameters. This controls the pressure between different chromosomes. Low number makes the pressure very strong from the beginning. High number: 0.99, or even 0.999, make the start close to random. Good choice is about 0.95 for easier problems with good building blocks and up to 0.999 for very hard problems
- `Give rank exponent [0.001000..10.000000]:`
  The second parameter for ranking selection. This determines how fast the pressure will change between different iterations. Bigger numbers make the pressure increase slowly. A good choice is between 0.5 and 2.
- `Do you want to dump rank samples to a file? [0..1]:`
  When ranking selection is used, you may dump simulated selection probabilities for all chromosomes to a file to visualize the resulting pressure. The file has a format suitable for spreadsheet loading. The simulation here is over an unrelated period (5o currently used). Enter 1 to select this option, and you will be prompted for a file name. Again, a new file will be created if the file exists.

# 9 CURRENT LIBRARY

Currently, only a FP vector representation is provided with **GenET** in `doubleVector`. This representation uses vector of double values and provides for bounded domains for the genes. In addition, a few sample operators are provided. It is our intension to collect a number of other operators for the same representation and make them available with future releases.

1. Current doubleVector representation definition – file `doubleVector/REP/representation.def`:
   a. there is no explicit definition for the chromosome so the file is empty,
   b. since this is an array of doubles, a double pointer is explicitly used whenever a reference to the actual chromosome is needed,
   c. the array size, or the number of genes, is provided by non-standard interface and is defined in `doubleVector/REP/representation.c`; it is called `NumGenes`.
2. In addition to `NumGenes`, the non-standard interface (header `doubleVector/REP/representationNS.h`) also provides lower and upper domain boundaries by arrays `LowerDom` and `UpperDom`. Both arrays are implemented in `doubleVector/REP/representation.c`.
3. Non-standard user interface provides the values for the number of genes and the domain boundaries.
4. Current operators – file `doubleVector/REP/operators.c` (all are closed in the domains):
   a. non-uniform mutation (#1), which dramatically denses the probability distribution for the new gene value around the current value as the population ages; it applies to only one gene per chromosome; its additional interactive parameter specifies the speed with which older populations increase the density – the higher the number, the higher the speed, best are up to 10; use this operator and higher numbers for problems requiring very high precision of genes,
   b. non-uniform mutation (#2), which denses the probability distribution for the new gene value around the current value as the population ages, but does it less dramatically than #1; it applies to only one gene per chromosome; it also uses one additional interactive parameter, which should best be -10..0 – the lower, the faster the change in density; again, use for high precision problems,
   c. a one-point crossover between genes,
   d. a uniform crossover (exchanging random genes); it uses one interactive parameter – the probability of swap; use 0.5 to have the actual uniform crossover,
   e. an arithmetical crossover averaging two whole chromosomes.

# 10 EXAMPLE APPLICATIONS

Two examples are provided:

1. A dynamic control problem from an early publication (see references in the file), which minimizes real parameters using the library representation `doubleVector`. It is implemented in `doubleVector/dynCntrl.c`.
2. A simple structure problem illustrating applications with implicit representation and heterogeneous genes. It is provided in `noLibProblems/simpleExample.c`.

# 11 USAGE AS PROGRAMMING TOOLBOX

**GenET** package offers expendability in many dimensions:

1. User-provided operators can include adaptable parameters, which may be made part of the representation or part of the operator.
2. Many different models can be simulated by the standard user interface.
3. Additional models/schemes can be easily implemented using the provided module interface. For example, roulettes may be created and spined (using interface in `ALG/roulette.h`), parents may be requested at will (using interface in `ALG/pop.h`), *etc*. A detailed programming manual will be made available in the future.

# 12 CURRENT LIMITATIONS AND PLANNED EXTENSIONS

Currently, the generational model and survival selection by roulette are not completed.

In addition to finishing these options, the proposed and being designed extensions include:

1. Implement additional scheme for operator probability adjustment which relies on offspring actual evaluations in addition to the current utility-based approach.
2. Provide extensive libraries of different representation and corresponding operators.
3. Provide tools to simplify construction of `problem.c` files.
4. Implement a steady-state mechanism where one operator is applied in a generation and the offspring either replaces or competes with its parent for survival