IMPLEMENTING SELC (SEQUENTIAL ELIMINATION OF LEVEL COMBINATIONS) FOR PRACTITIONERS —NEW STATISTICAL SOFWARES

by

TAN DING

(Under the Direction of Abhyuday Mandal)

ABSTRACT

Genetic algorithms (GAs) are a popular technology to search for an optimum in a large search space. Wu, Mao and Ma prosposed the sequential elimination of levels (SEL) method to find an optimal setting. Using the new ideas of a forbidden array and weighted mutation, Mandal, Wu and Johnson used elements of both SEL and GAs to introduce a new global optimization technique called sequential elimination of level combinations (SELC) to find optima more quickly. SELC has direct applications in pharmaceutical industries and hence, it is of importance to write statistical software to implement SELC automatically. In this thesis, we have implemented SELC in SAS, Matlab and R.

INDEX WORDS: Orthogonal array; Forbidden array; Weighted mutation; SAS; Matlab; R.

IMPLEMENTING SELC (SEQUENTIAL ELIMINATION OF LEVEL COMBINATIONS) FOR PRACTITIONERS – NEW STATISTICAL SOFTWARES

by

TAN DING

B.S., South China Normal University, China, 1999

M.S., Sun Yat-sen University, China, 2002

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment

of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2006

IMPLEMENTING SELC (SEQUENTIAL ELIMINATION OF LEVEL COMBINATIONS) FOR PRACTITIONERS – NEW STATISTICAL SOFTWARES

by

TAN DING

Major Professor: Abhyuday Mandal

Committee: Jaxk Reeves
Cheolwoo Park

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2006

ACKNOWLEDGEMENTS

It is my great pleasure to express my appreciation to those who have instructed, inspired, encouraged and supported me in my work of thesis.

First and foremost, it is my immense pleasure to express my deep and sincere gratitude to my advisor, Dr. Abhyuday Mandal, for his guidance, assistance, encouragement, and hearty support. His deterministic instructions were not only helpful for my thesis, but also for my future work.

I would also like to thank Dr. Kjell Johnson. Dr. Johnson gave me plenty of inspiration and guidance in implementing SELC in SAS and also helped writing my thesis.

I would also like to thank Dr. Jaxk Revees and Dr. Cheolwoo Park for their important advice and guidance.

Last but not the least, I would send my gratitude to my family. Their encouragement accompanies me throughout the whole process of my work.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

To search for an optimal solution in a large search space of potential candidates is a main goal in many scientific problems. As a popular optimization technique, Genetic Algorithms (GAs) are applied to select global optima from a large number of candidates. Additionally, a class of methods called SEL (sequential elimination of levels) is proposed to search for better settings. Sequential elimination of level combinations (SELC), which is a modification of SEL using the GA approach, can improve performance of GAs in several practical situations and can be used in some scenarios, especially pharmaceutical industries and computer experiments.

The main potential application of the SELC algorithm is in pharmaceutical industries. In the drug discovery process, work has primarily focused on finding chemical entities which are effective against a particular disease. The effective entities are generally obtained by chemical synthesis. Over the past decades, great technological progress has been made to explore and synthesize a large number of chemical compounds. This technology, known as combinatorial chemistry, has been widely applied in the pharmaceutical industry and is gaining interest in other areas of chemical manufacturing (Leach and Gillet 2003, Gasteiger and Engel 2003). Generally, combinatorial chemistry distinguishes molecules which can be easily combined together and physically makes each molecular combination by using robotics. In a synthesis process, the number of combinations can be extremely large when beginning with an appropriate number of molecules. For instance, theoretically one million potential products can be synthesized when 100 reagents are added to three locations onto a core molecule, since $(100)^3 = 1$ million.

Combinatorial chemistry has been used to enhance the diversity of compound libraries, to explore specific regions of chemical space (i.e., focused library design), and to optimize one or more pharmaceutical endpoints such as target efficacy or ADMET (absorption, distribution, metabolism, excretion, toxicology) properties (Rouhi 2003) in the pharmaceutical industry. In reality, it is not possible to follow up on each newly synthesized chemical compound, although it is possible to synthesize a large number of chemical entities in theory. Compared with synthesizing all possible molecular combinations, a better method is to computationally create and evaluate combinatorial libraries using structure based models. Additionally, a partial goal of chemists is to search for reagent combinations which produce undesirable compounds. Once such combinations are known, these combinations can be avoided during synthesis. Using these constraints, chemists select a subset of promising reagents to produce a combinatorial library. By construction, the SELC is an ideal method for searching for optimal molecules in technology of combinatorial chemistry.

The SELC method is also useful in computer experiments. Much research which in past decades could be conducted only by performing physical experiments can now be completed instead by computer experiments. In a computer experiment, a response, $y(x)$, is calculated for each set of input variables, x, using numerical methods implemented by (complex) computer codes (Santner, Williams, and Notz 2003). In such cases, the complex numerical methods can be thought of as a "black box" and the SELC algorithm can be applied to select the optima efficiently.

We can consider such real-life scenarios as large-dimensional design-of-experiment problems where the main challenge is to identify the optimal design settings. In scientific and engineering research, statistical design and analysis of experiments is an effective and commonly

used tool to understand and/or improve a system. Identifying important factors and choosing factor levels are among the first and most fundamental issues for an experimenter. But, when there is a large number of important factors, designing an experiment can be difficult. Classical experimental design relies heavily on algebraic properties such as orthogonality. However, orthogonality does not allow the flexibility to accommodate all kinds of promising follow-up runs, which, in turn, makes finding suitable designs for large-scale problems difficult, particularly when the factors have more than two levels.

The use of high-fidelity computer simulations of physical phenomena (Bates, Buke, Riccomagno, and Wynn 1996) has stimulated new research into ways in which experimental design can be applied to such problems. One technique, motivated by design of experiments, was introduced by Wu, Mao, and Ma (1990), and is known as sequential elimination of levels (SEL). The idea of SEL is opposite to that of the "greedy algorithm"; instead of focusing on factor levels that improve the response, SEL focuses on those levels that worsen the response. Based on this idea, SEL eliminates one level of each factor in each sequence of the experiment. However, SEL does not perform well in the presence of interactions. SELC, the modified version of SEL using the ideas from GAs will be used to improve the performance.

GAs have most often been viewed from a biological perspective. The metaphors of natural selection, cross breeding, and mutation have been helpful in providing a structure to explain how and why GAs work. Thus, most practical applications of GAs are rooted in the context where optimization is meaningful. In order to understand how GAs function as optimizers, Reeves and Wright (1999) took GAs to be a form of sequential experimental design. Recently, GAs have been applied successfully in solving statistical problems, particularly for searching for near-optimal designs (Hamada et al. 2001, Heredia-Langner et al. 2003, 2004) .

The thesis is organized as follows. Firstly, we review the GAs and the SEL in Chapter 2. Then, in Chapter 3 we propose SELC. We illustrate the procedure of SELC by an example. In the successive Chapters 4, 5 and 6, we introduce the implementation of SELC in SAS, in Matlab and in R software systems, respectively. Finally, we give the summary conclusions in Chapter 7.

CHAPTER 2

REVIEW: GENETIC ALGORITHMS AND SEQUENTIAL ELIMINATION OF LEVELS

Sequential elimination of levels combinations (SELC) is a modification of SEL using the idea from GAs. Before introducing SELC, we shall review GAs and SEL.

2.1 Genetic Algorithms (GAs)

Genetic algorithms, abbreviated as GAs, are a part of evolutionary computing, which is a rapidly growing area of artificial intelligence. GAs are a technique used in computing to find true or approximate optimal solutions. The method of GAs is inspired by Darwin's theory of evolution. As GAs are applied, the solving process to a problem evolves. GAs have found applications in computer science, engineering, economics, physics, mathematics and other fields.

Genetic Algorithms are implemented as a computer simulation in which a population of abstract representations (i.e., chromosomes) of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem evolves toward better solutions. Generally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population based on their fitness, and modified (recombined and possibly mutated) to form a new population. The new population is then used in the next iteration of the algorithm.

Genetic algorithms became a widely recognized optimization method as a result of the work of John Holland in the early 1970s, and particularly his 1975 book. His work originated

with studies of cellular automata, conducted by Holland and his colleagues at the University of Michigan. Research in GAs remained largely theoretical until the mid-1980s, when The First International Conference on Genetic Algorithms was held at the University of Illinois. As the academic interest grew, the dramatic increase in desktop computing allowed for application of the new technique in practice. In 1989, The New York Times writer John Markoff wrote about Evolver, the first commercially available desktop genetic algorithm. Custom computer applications began to emerge in a wide variety of fields, and these algorithms are now used by a majority of Fortune 500 companies to solve difficult scheduling, data fitting, trend spotting and budgeting problems, and virtually any other type of combinatorial optimization problem.

The basic idea of GAs is to solve problems by an evolutionary process which results in "better solutions" based on "good solutions".

The process steps of GAs are as follows:

1. Initialization.

Encoding is the first step for initialization of a problem with GAs. Encoding is also called solution representation. For problems requiring real number solutions, binary encoding is most commonly used where unique binary integers (e.g., 0 and 1) are mapped onto some range of the real line. Each bit is called a gene and a binary representation is called a chromosome.

There are many other ways of encoding. Encoding depends heavily on the problem. For ordering problems, permutation encoding is applied. Tree encoding is used mainly for evolving programs or expressions (i.e., genetic programming). Value encoding is a good choice for some special problems. After encoding, the GA proceeds as follows: Many candidate solutions are randomly generated to form a large

initial population. The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions.

Generally, the population is generated randomly, covering the entire range of possible solutions (i.e., the search space). Occasionally, the solutions may be "seeded" in areas where optimal solutions are likely to be found.

After initialization, the initial population is then continually transformed using Steps 2 and 3 described below.

2. Selection.

During each successive period, a proportion of the existing population is selected to breed a new generation. In the initial population, the individual best solution is selected and the worst one is eliminated on the basis of a fitness criterion (e.g., the larger, the better for a maximization problem). Certain selection methods rate the fitness of each solution and preferentially select the best solutions. Other methods rate only a random sample of the population, as this process may be very time-consuming. Most functions are stochastic and designed so that a small proportion of less fit solutions are selected.

This can help keep the diversity of the population large, preventing premature convergence on poor solutions. Popular and well-studied selection methods include roulette wheel selection and tournament selection.

3. Reproduction.

The purpose of reproduction is to transform the initial population into another set of solutions. For each new solution to be reproduced, a pair of “parent” solutions is selected for breeding from the pool of candidates. The step of reproduction is to generate a second generation population of solutions from those selected from the

initial population by applying the genetic operations called “crossover” and “mutation”:

a. Crossover.

A pair of binary integers (chromosomes) is split at a randomly chosen position, and the head of one is joined with the tail of the other and vice versa (Figure 1). Crossover operates on selected genes from parent chromosomes and creates new offspring.

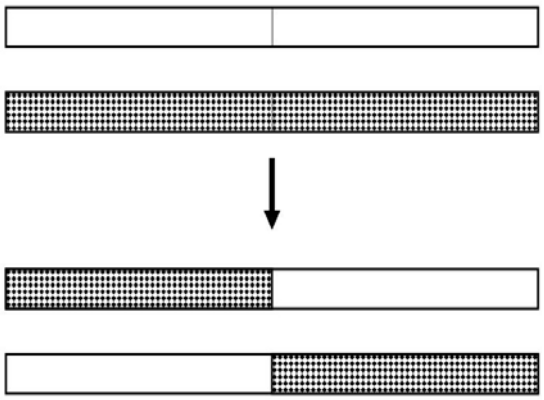

Figure 1: Crossover

b. Mutation.

The state (0 or 1) of a randomly chosen bit is changed (Figure 2). Mutation takes place after a crossover is performed. The intention of mutation is to prevent all

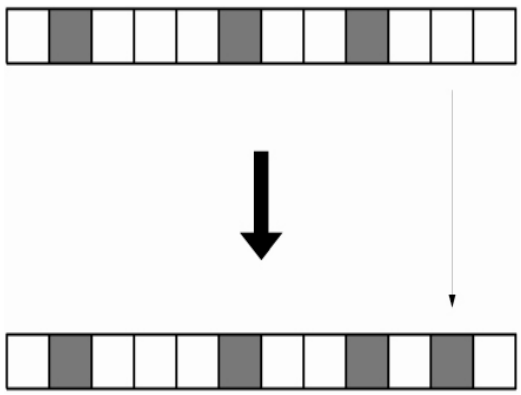

Figure 2: Mutation

solutions in the population from falling into a local optimum of the solved problem. By producing an "offspring" solution using the above methods of crossover and mutation, a new solution which typically shares many of the characteristics of its "parents" is created. New parents are selected for each child, and the process continues until a new population of solutions of appropriate size is generated.

These processes ultimately result in the next generation population of chromosomes that is different from the initial generation. Generally the average fitness will have increased by this procedure for the population, since only the best organisms from the first generation are selected for breeding, along with a small proportion of less fit solutions, for reasons already mentioned above.

4. Termination.

This generational process (i.e., Steps 2 and 3) is repeated until a termination condition has been reached. Common terminating conditions are:

a. fixed number of generations reached,

b. a solution that satisfies minimum criteria is found,

c. allocated budget (computation time/money) reached,

d. the highest ranking solution's fitness is reaching or has reached a plateau such that successive iterations no longer produce better results,

e. manual inspection,

f. combinations of the above.

2.2 Sequential Elimination of Levels (SEL)

Wu, Mao and Ma (1990) proposed sequential elimination of levels (SEL) as a method for searching over the orthogonal array to obtain better settings. The SEL method is motivated by

design of experiments. The procedure of SEL starts with an economically sized orthogonal array, with the search focusing on a smaller array on the remaining levels.

The SEL method includes the ideas of analysis of marginal means (ANOMM) and the pick-the-winner (PW) rule and starts with orthogonal array (OA). An orthogonal array OA (N, $s_1^{m_1}$, ..., $s_p^{m_p}$, t) of strength t is an N×m matrix, $m = m_1 + ... + m_p$, in which m_i columns have s_i (≥ 2) symbols or levels such that, for any t columns, all possible combinations of symbols appear equally often in the matrix. For OA of strength two, the index $t = 2$ is dropped for simplicity. For an example, here is an orthogonal array of strength 2 as below.

0 0 0 0

0 1 1 2

0 2 2 1

1 0 1 1

1 1 2 0

1 2 0 2

2 0 2 2

2 1 0 1

2 2 1 0

To see that it is orthogonal of order two, take any two columns (say the first and fourth ones) and one will have the sequence below:

0 0

0 2

0 1

1 1

1 0

1 2

2 2

2 1

2 0

One can then not that each of the 9 combinations such as {(0, 0), (0, 1), ..., (2, 2)} occurs equally often (once). The same condition can be verified for any of the six possible pair of columns.

The SEL algorithm is proposed as follows:

1. Begin with an appropriate orthogonal array and compute the responses.
2. For each factor, eliminate the level(s) with the worst mean value(s) of the performance measure computed from the current array.
3. Select an orthogonal array (of a smaller size) for the remaining levels, and replace the array in step 1 with the new one.
4. Conduct another experiment on the new array.
5. Repeat Steps 2-4 if necessary.

In Steps 1 and 2, SEL is call SEL(x) if the mean is replaced by another descriptive statistic x (e.g., minimum).

SEL is a very general method as it includes both ANOMM and PW as extreme cases. In Step2, if only one level survives the eliminations, SEL (means) reduces to ANOMM and SEL (mini) reduces to PW.

The SEL method searches for optima too restrictively, which is its main drawback. First, the SEL method is not optimal for the experiments containing important interactions, because the

algorithm eliminates individual levels of each factor without considering interactions. Thus, SEL can blindly eliminate a factor level required for the optimal run of the experiment. Second, SEL requires that subsequent experiments follow an orthogonal array. As mentioned previously, the SELC can prevent it from using an orthogonal array. Additionally, orthogonal arrays are not flexible enough to deal with complex response surfaces. To overcome this problem, Mandal et al. (2006) have developed sequential elimination of level combinations (SELC) to determine subsequent design points.

CHAPTER 3

A BRIEF ACCOUNT OF SELC ALGORITHM

The sequential elimination of level combinations (SELC) is a global optimization technique which is motivated by genetic algorithms but finds the optimum more quickly by using novel ideas from design of experiment literature.

In SEL, Wu et al. eliminated individual levels. In SELC, the level combinations are eliminated. Instead of one factor at a time, the factor settings, which have the same level combinations as that of the worst setting for two factors, are eliminated. When dimensions are large, we may have to consider the third or higher-order tuples to narrow down the search space. The worst captured runs are placed in the forbidden array as the searching process continues. After putting worse runs in forbidden array, it uses GAs to find new runs. Following GAs, the SELC algorithm suggests runs for new experiments, and better runs from previous experiments are used to produce promising settings for a new run.

The two key features of the SELC algorithm, namely, forbidden array and weighted mutation, enhance the performance of the search procedure. SELC starts with an orthogonal design that helps in identifying important effects. The follow-up runs are very flexible and data-driven; the weighted mutation uses sequential learning. This SELC method is useful in many real-life examples, ranging from computer experiments to compound identification in pharmaceutical industries.

For example, consider a core molecule in pharmaceutical industry, for which there are four locations on each of which three possible reagents can be added (Figure 3). An experimenter

needs to search for a new entity (with 3 reagents added onto 4 locations of core molecule) which is the most effective against a particular disease. It would be complex to synthesize all the compounds because there are totally 81 (=3^4) combinations in theory. Applying SELC can make the problem easy. Take each location as a factor and let the three possible reagents be thought of as three levels. Then, the problem is now to find an optimal level combination in a large number of candidates. This example will be used later to illustrate the SELC process.

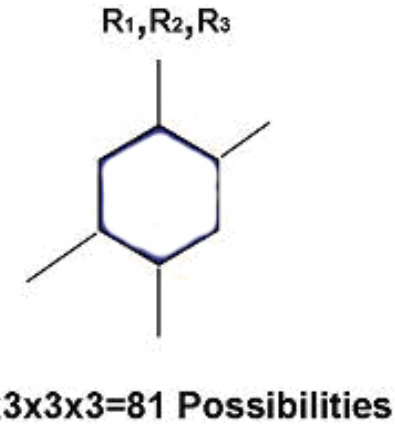

Figure 3: Hypothetical Example from Pharmaceutical Industry

Before introducing SELC algorithm, we define the important concepts of forbidden array and weighted mutation, which are both necessary for the algorithm.

3.1 Forbidden array

One of the key features of the SELC method is the idea of forbidden array. Sometimes, prior knowledge or experience is available and the knowledge suggests that some specific factor-level combinations will generate undesirable results. In this situation, we can place these specific combinations into the forbidden array before initializing the SELC algorithm.

When prior knowledge is unavailable, the SELC algorithm will be initialized with an orthogonal design. The initial experiment evaluates setting(s) which are not optimal. These worst setting(s) are then placed into the forbidden array.

First, we select the worst run(s) with probability governed by a "fitness" measure (i.e., value of response) and store the run(s) in the forbidden array. Furthermore, two features of the forbidden array, strength and order should be specified. Strength is defined as the number of runs stored into the forbidden array. More specifically, a forbidden array with strength *s* consists of the *s* worst runs of the experiment at each stage of the iterations. Additionally, the runs in the forbidden array define a set of level combinations which are forbidden in the experiment. Order is defined as the number of level combinations which are banned from subsequent experiment in forbidden array. A forbidden array with order *k* means that any combination of *k* or more levels from any array in the forbidden array will be banned in the experiment. Thus, the smaller is the order, the larger is the number of forbidden design points. Consequently, the forbidden array is a set in which all runs are forbidden by SELC.

For instance, suppose there is an experiment in which the goal is to maximize a response. Assume that there are four factors, each has three levels (0, 1 and 2), and we construct a forbidden array with strength 1 and order 2. Further suppose that we obtain the minimum value of E(y) when all factors are set to be 0, and this design point is run during the experiment. When we store this run into the forbidden array, this run will prohibit any design point with two or more zero-value factors (order=2). Note that in this case, there is only one run stored in the forbidden array (strength=1).

We use an example to illustrate the construction of forbidden array. Recall the previous example from pharmaceutical industry. Assume that there are four positions, each of which can be filled with three reagents. The response (efficiency of the compound) is computed by the following model:

$y = 40 + 3A + 16B - 4B^2 - 5C + 6D - D^2 + 2AB - 3BD + \varepsilon,$ where ε is a standard normal error.

We have an orthogonal array of four factors and each factor has three levels and appropriate responses to each factor level combination (Table 1).

Table 1: OA ($9, 3^4$) and y

A	*B*	*C*	*D*	*y*
1	1	1	1	10.07
1	2	2	3	53.62
1	3	3	2	43.84
2	1	2	2	13.40
2	2	3	1	46.99
2	3	1	3	55.10
3	1	3	3	5.70
3	2	1	2	43.65
3	3	2	1	47.01

In this case, the aim is to find a setting which has maximum y value. Firstly, we choose the "worst" combination as follow:

A	*B*	*C*	*D*	*y*
3	1	3	3	5.70

Forbidden array consists of rums with the same level combinations as that of the "worst" one at any two positions (Table 2).

Table 2: Forbidden Array

A	*B*	*C*	*D*
3	1	*	*
3	*	3	*
3	*	*	3
*	1	3	*
*	1	*	3
*	*	3	3

where * is the wildcard which represents any admissible value.

Once the forbidden array is constructed, SELC begins to search for better level settings. The search procedure is motivated by GAs. The first step is encoding (Section 2.1). The runs are viewed as choromosomes. For a *k*-level factor, the levels can be denoted by 0, 1, .., *k*-1. For

instance, for a 3^4 (4 factors, each has 3 levels) experiment, the design points (i.e., chromosomes) can have the form (0, 0, 0, 0), (1, 0, 0, 0), ..., (2, 2, 2, 2). Encoding here is a modification of binary arrays. After encoding chromosomes, we identify the best runs for generating offspring of the next generation with probability proportional to the "fitness" (i.e., y value). The chosen good candidates reproduce to generate potential better candidates. In SELC, crossover is conducted the same way as it is done in classic GAs (chapter 2) while the mutation step is modified.

3.2 Weighted mutation

In the mutation step of GAs, genes mutate with equal probabilities. This mutation process does not use gathered information based on previous knowledge. As we discussed with SELC, mutation probabilities can be calculated by using prior information. For example, a specific factor "*F*" has a significant main effect and no significant two-factor interactions. Then we will mutate for this factor with probability p_i, where

$$p_i \propto \bar{y}(F = i), \qquad (1)$$

where $\bar{y}(F = i)$ stands for the average value of y corresponding to the level i of factor F.

Next, the effect of interaction of factors F_1 and F_2 is significant. Then, the mutation probability should be a joint probability of F_1 and F_2. When either F_1 or F_2 is randomly chosen, the mutation will be weighted mutation. Factor F_1 will be set to i_1 and factor F_2 will be set to level i_2 with probability $q_{i_1 i_2}$ where

$$q_{i_1 i_2} \propto \bar{y}(F_1 = i_1, F_2 = i_2). \qquad (2)$$

If the chosen factor does not have significant main or interaction effects, then its value will be changed to any possible levels with equal probability. This is called normal mutation.

Generally, we use a linear regression model to identify the significant effects. A Bayesian variable selection strategy may be a better approach.

3.3 Starting design

We will begin the SELC with an orthogonal array which helps to estimate factor effects efficiently. The information on factor effects would be used in the process of weighted mutation.

3.4 The SELC Algorithm

Begin with an initial design (i.e., an orthogonal array)

1. Conduct the experiment.

   -Stop when the stopping criterion is achieved (see later).

2. Construct the forbidden array and specify appropriate strength and order.

3. Generate *b* new offspring.

   -Select offspring with probability proportional to their "fitness".

   -Crossover.

   -Mutation. Using weighted mutation.

4. Check eligibility and novelty. An offspring is eligible if it matches: 1) new offspring is not prohibited by any of the elements of the scientific forbidden array; 2) New offspring is not prohibited by any of the elements of the forbidden array which is constructed in Step 2; 3) New offspring is different from any of the elements of the initial design. If an offspring is ineligible, then generate a new offspring and abandon the ineligible one.

5. Repeat Steps 3, 4 and then stop when *b* new eligible offspring have been created. If *b*=1 and more than one offspring were generated, then randomly select one for the experiment.

The SELC method can be either full (*b*=1) or batch (*b*=*b*, where *b*>1) sequential in different situations. For fully sequential SELC, a new eligible offspring is produced each

iteration and the experiment is conducted. For batch sequential SELC, a new set of eligible offspring is produced each iteration and the experiment is conducted.

3.5 Stopping rules

The stopping rule is subjective and depends on progression of the algorithm and experimental constraints.

As the runs are added one by one, the experimenter can decide whether a significant progress towards optimization has been made and can stop after near optimal solution is obtained. Sometimes, there is a predetermined value for the experiment. Once the target value is obtained, the search process can be stopped. Most frequently, experiments stop because the number of experiments is limited by the resources at hand.

3.6 Examples

We use the previous pharmaceutical industry's example to illustrate how the SELC algorithm works. We have already constructed a forbidden array with *strength*=1 and *order*=2 (Table 2). The SELC starts to search for better level settings. The search procedure is motivated by GAs. The best two runs denoted by P_1 and P_2 are chosen as follows:

	A	B	C	D	y
P_1	2	3	1	3	55.10
P_2	1	2	2	3	55.62

Randomly select a gap from *A*-*B, B*-*C* or *C*-*D*, (say, *B*-*C*) and do crossover at this position. Two offspring denoted by O_1 and O_2 are thus generated as follows:

	A	B	C	D
O_1	2	3	2	3
O_2	1	2	1	3

In this case, a Bayesian variable selection strategy is employed to identify the significant effects. One can also use linear and/or quadratic regression to identify two significant effects. In this case, effects B and B^2 turn out to be significant.

For factors which possess significant main effects, we evaluate p_i in model (1) (Section 3.2), and for factors which have significant interaction effects, we must also evaluate the $q_{i_1 i_2}$'s in model (2) (Section 3.2).

If factor B is randomly selected for mutation, then we calculate mutation probabilities for each level as follows:

$$p_1=0.09,\ p_2=0.45,\ p_3=0.46.$$

These values arise from the significant weighting. If a factor which is insignificant is selected, its mutation probabilities are uniformly distributed over the available levels.

In this simulation, for O_1, factor A is selected for mutation and the level is changed from 2 to 1. For O_2, factor B is chosen and the level is changed from 2 to 3. The new offspring are:

	A	B	C	D
O_1	1	3	2	3
O_2	1	3	1	3

Neither of the two offspring are prohibited by the forbidden array (Table 2), so both are eligible and are "new" level combinations (i.e., the offspring are different from any run in the initial design).

CHAPTER 4

IMPLEMENTING SELC IN SAS

The SAS system is a comprehensive software package with very powerful tools of data management, and a wide variety of statistical analysis and many graphical procedures. Because of its versatility, SAS is widely used in academic research and industry to deal with data. This "SELC" SAS package is convenient and easy for users to conduct SELC algorithm. Using the tools in SAS, we have constructed a macro to implement the SELC algorithm. The "SELC" macro has a number of components and employs several modules and procedures which we describe below.

4.1 SAS Components Used in the "SELC" Package

A large computer software system, the SAS system consists of a number of components. The "SELC" macro applies the SAS/IML, SAS/BASE and SAS/STAT modules.

The SAS/BASE is the core of the SAS system. This component is used to manage data. SAS/BASE includes many procedures, the Macro facility and Output Delivery System (ODS), which are applied to implement SELC.

The SAS/STAT is powerful to conduct statistical analysis with a number of procedures, providing statistical information.

To implement the SELC algorithm, it is more convenient to treat the data as matrix for steps such as crossover, mutation and eligibility. In SAS/BASE and SAS/STAT, data are treated as vectors and processing row by row. When dealing with the whole data matrix, we need to employ SAS/IML.

The SAS/IML is a powerful and flexible programming language (Interactive Matrix Language) in SAS system. Unlike most programming languages which deal with single data elements, SAS/IML software's fundamental data element is matrix.

4.2 Special SAS Tools Used in the "SELC" Package

In SAS system, there are many internal tools. Several tools introduced as follows are helpful in implementing SELC.

1. Output Delivery System (ODS).

   Output Delivery System (ODS) is a system that delivers output in a variety of formats. Generally, the output of a procedure includes complex information. If only partial output or just a value of a procedure is needed, ODS can carry out such delivery.

   The syntax of ODS is as follows:

```
ODS OUTPUT GoalFormat=OutputName;
ODS LISTING CLOSE;
Procedure;
ODS LISTING;.
```

   In the first line, the goal format is specified and put into a data set name (OutputName). ODS LISTING CLOSE clause closes the ODS temporarily. Common SAS procedures can be performed next and the needed information will be stored in a data set "OutputName". At last, the ODS LISTING clause creates a data set storing the aimed information which will not appear in the SAS Output Window. Then, the data set "OutputName" includes useful information and can be called in next procedures.

   In the "SELC" macro, we use ODS when we check the significant effects. The information on significant factors which are produced by PROC GLM statements is saved in an internal data set.

2. MACRO facility.

The macro facility is a tool for extending and customizing SAS software programs and reducing overall program verbosity. It can be used to pack many procedures to perform a particular process.

The entire macro syntax is as follows:

```
%MACRO macroname(VAR1=, VAR2=, …);
   procedures and statements;
%MEND macroname;
%macroname (VAR1=value1, VAR2=value2, …);.
```

The first clause sets a macro with arguments (e.g., VAR1, VAR2, ...). The macro ends at the %MEND clause. The macro will not work until the arguments are set values. In the clause %macroname, user can set fixed values to all the arguments.

The "SELC" package consists of three macros; we introduce these three macros in Section 4.3.

3. Defining Macro Variables.

In implementing a process, many macro variables are needed. A macro variable in SAS is a string variable that allows user to dynamically modify the text in a SAS program through symbolic substitution.

Macro variables can be specified in the call to the macro. More simply, macro variables can be defined using the %LET statement. The syntax is:

```
%LET VariableName=value;.
```

By %LET statement, the macro variable (e.g., VariableName) can only be given a fixed value. In SELC macro, %LET statement is used to input true level of each factor in the SELC statement. The statement can input value directly.

In the particular case, the following codes are example of %LET statement:

```
%LET L1=x;
%LET L2=y;
  .
  .
  .
%GLOBAL  L1 L2 …;.
```

where the variable names denote levels for factor 1, factor2,....The "%GLOBAL" clause makes these variable eligible in the entire macro.

Macro variables need to be defined spontaneously based on characteristics of the data. There are several ways to generate this information using SAS. We will focus on the SQL procedure and the CALL SYMPUT statement.

The SQL procedure (PROC SQL) is a wonderful tool for summarizing (or aggregating) data. It provides a number of useful summary (or aggregate) functions to help perform calculations, descriptive statistics, and other aggregating operations in a SELECT clause or HAVING clause. These functions are designed to summarize information and not display detail about data.

The SQL procedure syntax is as follows:

```
PROC SQL;
     SELECT function(var)
     INTO: MacroVar
     FROM dataset;
QUIT;.
```

The CALL SYMPUT statement is another way of generating macro variable spontaneously. This statement can be used only in an automatically Data statement.

The syntax is as follows:

```
DATA _NULL_;
     functions;
     CALL SYMPUT ('macrovar', var);
RUN;
```

Then the macro variable "macrovar" is created.

By using SQL procedure or CALL SYMPUT statement, the macro variables are generated in genetic process and have changeable values.

4.3 Macros and Arguments

In this "SELC" package consists of 3 macros. Macro "SELC" implements SELC using information created in macros "SIGCHECK" and "MUTPROB".

1. Macro "SIGCHECK".

   This macro uses a linear regression model to check significant effect(s). No quadratic effect or interaction effect is considered.

2. Macro "MUTPROB".

   In this macro, mutation probability for a factor is calculated. If the factor is significant, then the probability is weighted mutation probability. If the factor is insignificant, then the probability is equivalent probability.

3. Macro "SELC".

   This marco implements entire process of SELC algorithm. Useful information created in Macro "SIGCHECK" and Macro "MUTPROB" will be used in this macro.

To carry out macros, a user needs to give values to all arguments in the "%SELC" clause which is below the "SELC" macro. In the "SELC" macro, the arguments need to be specified are as follows:

1. dfile.

   This is initial data set including factors and response, successively. The path of the file should be pointed out at the same time when the data set is specified.

2. ffile.

   This is forbidden array based on prior knowledge. Same as dfile argument, both data set and path should be inputted.

3. numoff.

   This is the required number of offspring.

4. strength.

   This is strength of forbidden array. Strength is a natural number.

5. order.

   This is order of forbidden array. Order is a natural number smaller than the number of factors.

6. numfact.

   This is the number of factors in initial data.

7. dir.

   This argument is related to fitness criterion. If maximum is preferred, “dir” equals 1 while if minimum is desired, “dir” equals 0.

8. wt.

   This argument is to create y-related values used in choosing parent runs.

9. alpha..

   This is level of significance. When check significant effects, the effect whose p-value is smaller than alpha is significant. Generally, alpha is 0.05.

Unlike the above arguments, true levels need to be input in the "SELC" macro. True levels are the possible levels for each factor. For example, there is one factor *A*. *A* has totally seven levels: 1, 2, 3, 4, 5, 6, 7 and only four levels 1, 2, 4 and 7 exist in a data set. The true level of factor *A* in this data set is seven but not four.

4.4 Example

An example from combinatorial chemistry (Figure 4) is used to illustrate how the "SELC" software works. On a core molecule, three locations (denoted by A, B and C) can be added by reagents. For location A, five reagents can be added. Thirty-four reagents are available for location B and two hundred and forty-one are available for location C. The goal is to find the most powerful compound with three reagents filled onto the three locations on the core molecule. This example can be translated as that for a three-factor model, to find an optimal solution or level combination which produces optimal response. The possible levels for three factors *A, B* and *C* are 5, 34 and 241 respectively. Then, using the SELC algorithm, we can solve the problem easily.

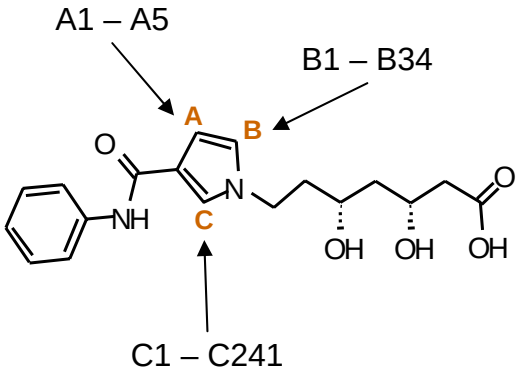

Figure 4: Hypothetical Example from Combinatorial Chemistry

We start with an initial design (see Appendix A1) and have a given 71–run forbidden array (see Appendix A2). First, at the bottom of the program, we input the possible levels for each factor using %LET statement as follow:

```
%LET L1=5;
%LET L2=34;
%LET L3=241;
%GLOBAL L1 L2 L3;.
```

After that, we specify the arguments in %SELC statement which is the last line of the program as:

```
%SELC(dfile="c:\intialdesign.txt",
      ffile="c:\forbiddenarray.txt",
      numoff=10,
      strength=2,
      order=2,
      numfact=3,
      dir=1,
      wt=1,
      alpha=0.05);.
```

We set initial design and given forbidden array ready (`dfile="c:\intialdesign.txt", ffile="c:\forbiddenarray.txt"`) and specify strength and order (`strength=2, order=2`). In this example, we need 10 offspring (`numoff=10`). In this case, there are 3 factors (`numfact=3`) and maximum is preferred (`dir=1`). That "wt=1" tells program choose "parent" offspring not so sensitive and that "alpha=0.05" means that those effects with p-values smaller than alpha are significant.

In this example, we require 10 offspring and the SELC package generates 10 eligible offspring as follows:

Obs	x1	x2	x3
1	1	6	2
2	1	21	6
3	2	6	40
4	2	9	227

5	2	15	5
6	2	19	30
7	4	4	35
8	4	16	30
9	4	24	11
10	5	15	40

CHAPTER 5

IMPLEMENTING SELC IN MATLAB

Matlab allows easy matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages. As it is described previously (Chapter 4), some steps of SELC such as crossover, mutation and eligibility can be implemented easily in matrix mode, so implementing SELC in Matlab is relatively easy. Furthermore, Matlab is more powerful than SAS to control data, so the Matlab program has more complex function (when check significant effects, matlab employs a quadratic model) than SAS does.

5.1 Functions or Module Available in the "SELC" Package

The "SELC" Matlab package is composed of 12 modules as follows:

1. SELC.

   This function is the only function which will produce the desire offspring as output. In the function, many useful variables are created by other functions described as following.

2. evalsig/modelmatrix (module).

   This module has two functions. One constructs a design matrix and the other checks significant effects using a quadratic regression model.

3. choose_with_prob.

   Parents for reproduction are selected with probability proportional to x value by this function.

4. crossover.

   The function invokes crossover.

5. is_there_imp_2fi.

   This function checks significant interaction effects.

6. identify.

   Given the mutation location, this function help identify the parent factors. For example, if there are 4 factors, *A, B, C* and *D* and we have known that *A* and *AD* are important, then, the function will produce a value "i" which results in a returned value of factor.

   If i=1, function will return *A* and *D* (D will come as AD interaction is significant);

   If i=2, function will return *B*;

   If i=3, function will return *C*;

   If i=4, function will return *A* and *D* (*A* will come as *AD* interaction is significant).

   The "i" will be used to decide one-factor weighted mutation or two-factor weighted mutation should be conducted.

7. prob_main_effect.

   The function helps compute mutation probability of main effect for factor.

8. one_factor_mutation.

   This function performs one-factor mutation.

9. prob_2fi.

   This function computes the two-factor mutation probabilities.

10. two_factor_mutation.

    This function carries out two-factor mutation.

11. mutation.

    Mutation step will be conducted by this function. This function includes function "one_factor_mutation" and function "two_factor_mutation". If randomly selected factor for mutation is insignificant, then the function performs normal mutation. If the factor chosen has only significant main effect, then the mutation will be on that effect, while if the factor has significant interaction effect, mutation will be joint on both of the related factors.

12. check.

    After mutation, we check the eligibility. The new offspring is eligible if it is not prohibited by any setting in initial forbidden array and in the forbidden array created without prior knowledge and it different from any existing run in initial design. Eligible offspring will be stored in data set "newruns".

5.2 Arguments

In the 'SELC' package, function "SELC" is the only function for which a user needs to specify the initial design, forbidden array based on prior knowledge (if available) and some arguments.

These arguments are:

1. strength.

   This is strength of forbidden array. Strength is number of runs to be stored in forbidden array based on initial design.

2. order.

   This is order of forbidden array. Order should be a natural number not larger than number of factors.

3. number_of_offspring.

   This is the required number of new offspring.

4. level_true.

   This should be a vector which consists of all possible levels of all factors despite absence of some levels. For example, factor *A* has totally 7 levels in which only 5 levels present in initial design and factor *B* has totally 10 levels in which only 9 levels present in initial design, then the level_true should be (7 10).

5.3 Example

Recall the example from combinatorial chemistry (in Section 4.4). We use the example to illustrate the process of SELC in Matlab. Here, a step-by-step illustration and a last output will be conducted.

First of all, set the initial design and the forbidden array data sets ready.

```
>> initialdesign = textread('initialdesign.txt');
>> forbiddenarray = textread('forbiddenarray.txt');
```

Then we specify several arguments.

```
>> strength=2 ;
>> order=2 ;
>> number_of_offspring=10 ;
>> level_true=[5 34 241] ;
```

Applying the function ‘evalsig’, the significant effects of *B* and B^2 are checked. According to the check result, if *B* is randomly selected for mutation, do one-factor mutation.

```
>> sigfactor = evalsig(yc,x);
>> sigfactor
   sigfactor =
        0     1     0     0     1     0     0     0     0
```

Function ‘choose_with_prob’ helps to select 2 parents denoted as p1 and p2. After that, p1 and p2 are crossovered and mutated to produce offspring p1new and p2new.

```
>>candidates = choose_with_prob(y,population_size);
      p1 = x(candidates(1),:);
      p2 = x(candidates(2),:);
      [p1new p2new] = crossover(p1,p2);
      p1new =
mutation(x,y,sigfactor,p1new,num_main_effects,level_true);
      p2new =
mutation(x,y,sigfactor,p2new,num_main_effects,level_true);
   >> p1
        p1 =
            3     4   172
   >> p2
        p2 =
            3     8    40
   >> p1new
        p1new =
            3     4   111
   >> p2new
        p2new =
            2     8   172
   >> indicator3
        indicator3 =
            1
   >> indicator4
        indicator4 =
            1
```

After checking eligibility using function “check”, we know that offspring, p1new and p2new, are both eligible, so SELC algorithm generates 2 offspring and they are placed in a set “newruns”.

In this case, argument “number_of_offspring” is set to 10. The SELC function will then generate 10 offspring as follows:

```
>> newruns

newruns =

     4    15    14
     4    28    11
     1    15   113
     3    29     6
     2    13     5
     3     4    92
     3    11    14
     1    28   172
     2    10    13
     3    30    10
```

CHAPTER 6

IMPLEMENTING SELC IN R

The R language is a programming language and software environment for statistical computing and graphics. As free software, the R is widely used for statistical programming development and data analysis. Similar as Matlab, R is powerful when dealing with data both as vector or matrix.

6.1 Functions in the "SELC" R package

This "SELC" R package is composed of 13 functions. Unlike Matlab program, functions cannot be packed in a module and the order of functions is important in R program.

We introduce the functions in order as follows:

1. modelmatrix.

   The function constructs the design matrix which will be used in a regression model.

2. evalsig.

   This function uses a regression model to check significant effects, both main effect and interaction effects. The design matrix constructed by function "modelmatrix" is used.

3. choose_with_prob.

   Two offspring are selected for reproduction with probability proportional to y value by this function.

4. crossover.

   The function applies crossover.

5. is_there_imp_2fi.

   When significant interaction effects are checked, the two factors having significant interaction should be fixed. This function will complete this task.

6. identify.

   Given the mutation location, this function identifies the parent factors.

7. prob_main_effect.

   This function helps calculate mutation probability for main effect of a factor.

8. one_factor_mutation.

   A one-factor mutation occurs by this function.

9. prob_2fi.

   This function helps calculate mutation probabilities for two factors.

10. two_factor_mutation.

    When an interaction effect is significant, we consider a two-factor mutation. The two-factor mutation is completed by function.

11. mutation.

    This function will complete mutation operation. The information created in the previous described functions “one_factor_mutation” and “two_factor_mutation” are both included in this function. If an insignificant factor is chosen to be mutated, then conduct mutation with an equivalent mutation probability, otherwise mutation with an appropriate weighted mutation probability will be conducted when the chosen factor is significant.

12. check.

    In this function, generated offspring will be checked for eligibility.

13. SELC.

This function generates offspring after all previous functions set. In this package, the function “SELC” is at the bottom of the program.

6.2 Arguments

To apply the SELC program, user should set the initial design and forbidden array and specify some arguments before running the program. The arguments are:

1. strength.

   Strength of forbidden array.

2. order.

   Order of forbidden arrary.

3. level_true.

   For each factor, user should specify the true level. Generally, in initial design, not all levels for all factors exist, so to specify the true levels is very necessary for mutation step.

4. number_of_offspring.

   This variable tells the package how many offspring user needs.

6.3 Example

We use the exampl in Section 4.4 to illustrate the process of SELC in R.

Firstly, we set initial data and forbidden array ready and specify several arguments.

```
> initialdesign=read.table(“c:/Initialdesign.txt”)
> forbiddenarray=read.table(“c:/forbiddenarray.txt”)
> initialdesign=as.matrix(initialdesign)
> forbiddenarray=as.matrix(forbiddenarray)
> strength=2
```

```
> order=2
> level_true=c(5,34,241)
> number_of_offspring=10
```

In this case, a non-intercept regression model to check the significant effects. The result is that linear effect B, quadratic effect B^2 are significant. According to the result, factor B should be weighted mutated.

```
> sigfactor = evalsig(x,yc)
> sigfactor
   0    1    0    0    1    0    0    0    0
```

The function "choose_with_prob" helps choose two parents runs, p1 and p2.

```
>     p1 = x[candidates[1],]
>     p2 = x[candidates[2],]
> p1
          1  6 32
> p2
          4 22  6
```

In crossover step, two offspring generated. The new offspring are p1new and p2new. Here, the randomly selected crossover location is at factor B's position.

```
>          newoff = crossover(p1,p2)
>          p1new = newoff[1,]
>          p2new = newoff[2,]
> p1new
          4 22  6
> p2new
          1 6 32
```

After crossover, mutation is conducted. In this case, factor C is mutated.

```
> p1new = mutation(x,y,sigfactor,p1new,num_main_effects,level_true)
```

```
> p2new = mutation(x,y,sigfactor,p2new,num_main_effects,level_true)
> p1new
            4  8  6
> p2new
            1 6 100
```

At last, "check" function checks the eligibility for each offspring. In this particular case, two offspring, p1new and p2new, are both eligible and will be stored in a matrix named "newruns".

```
> newruns
           p1new  4  8   6
           p2new  1  6 100
```

At the beginning of this example we specify value of "number_of_offspring" as 10, so 10 eligible offspring are generated after running the entire program.

```
> SELC(strength, order, level_true, number_of_offspring)
     p1new  2 15  22
     p2new  3 22 218
     p1new  4  7 129
     p2new  1  6 199
     p1new  4 28 186
     p2new  3 15  46
     p1new  4  9  32
     p2new  1  6  25
     p1new  3  9 227
     p2new  3 28  99
```

CHAPTER 7

SUMMARY AND CONCLUSIONS

In this thesis, we discuss an optimization technique: sequential elimination of level combinations (SELC) algorithm and implement SELC in the SAS, Matlab and R software systems, respectively.

To select the optimal design setting in a large number of candidates is not easy. The SELC method can conduct the searching task efficiently. The SELC algorithm is a modification of SEL using the idea of GAs. SELC considers important interaction effects besides significant main effects. This consideration reduces the probability of blindly eliminating levels and enhances the likelihood of obtaining the best setting in smaller runs. Furthermore, starting with an economically sized orthogonal array and using the novel ideas of forbidden array and weighted mutation can make the search process more efficient. The SELC algorithm can be very useful to the pharmaceutical industry.

R is free software, whereas SAS and Matlab are popular in pharmaceutical industry, so, in the thesis, we wrote programs to help practitioners implement SELC automatically using all three of these software packages.

REFERENCES

Bates, R. A., Buke, R. J., Riccomagno, E., and Wynn, H. P. (1996), Experimental Design and Observation for Large Systems. Journal of the Royal Statistical Society, Ser. B, 58, 77-94.

Delwiche, L. D., and Slaughter, S. J. (2003). The Little SAS Book: Third Edition. Cary, NC: SAS Institute Inc..

Dilorio, F. C., and Hardy, K. A. (1996). Quick Start To Data Analysis With SAS. Belmont, CA: Duxbury Press.

Dixon, L. C. W., and Szego, G. P. (1978). The Global Optimization Problem: An Introduction In Towards Global Optimization 2 (Edited by L. C. W. Dixon and G. P. Szego), 1-15. Amsterdam, Netherlands.

Gasteiger, J. and Engel, T. (eds.) (2003), Chemoinformatics : a Textbook. Weinheim, Germany: Wiley-VCH.

George, E. I., and McCulloch, R. E. (1993). Variable Selection Via Gibbs Sampling. Journal of the American Statistical Association. 88, 881-889.

Hedayat, A. S., Sloane, N. J. A., and Stufken, J. (1999), Orthogonal Arrays : Theory and Applications. New York, NY: Springer-Verlag Inc..

Heredia-Langner, A., Carlyle, W.M., Montgomery, D.C., Borror, C.M. and Runger, G.C. (2003). Genetic Algorithms for the Construction of D-optimal Designs. Journal of Quality Technology 35, 28-46.

Heredia-Langner, A., Montgomery, D. C., Carlyle, W. M., Borror, C. M. (2004). Model-

Robust Optimal Designs: A Genetic Algorithm Approach. Journal of Quality Technology, 36, 3, 263-279.

Holland, J. M. (1975). Adaptation in Natural and Artificial Systems. Ann Arbor, MI: The University of Michigan Press,.

Holland, J. M. (1992), Adaptation in Natural and Artificial Systems. Cambridge, MA: MIT Press.

Hunt, B. R., Lipsman, R. L., Rosenberg, J. M., Coombes, K. R., Osborn, J. E., and Stuck, G. (2006). A Guide to MATLAB: for beginners and experienced users: Second Edition. Cambridge, UK: Cambridge University Press.

Leach, A. R., and Gillet, V. J. (2003), An Introduction to Chemoinformatics. London, UK: Kluwer Academic Publishers.

Maindonald, J. H., and Braun, J. (2003). Data Analysis and Graphics Using R. Cambridge, UK: Cambridge University Press.

Mandal, A. , Wu, C. F. J., and Johnson, K. (2006). SELC: Sequetial Elimination of Level Combinations by Means of Modified Genetic Algorithms. Technometrics, 48, 273-283.

Pratap, R. (2006). Getting Started with MATLAB 7: A Quick Introduction for Scientists and Engineers. New York, NY: Oxford University Press.

Reeves, C. L., and Wright, C. C. (1999). Genetic Algorithms and the Design of Experiments. In Davis, L. D.; DeJong, K.; Vose, M. D. and Whitley, L. D. (1999) (Ed.). New York, NY: Springer-Verlag Inc., pp 207-226.

Rouhi, A. M. (2003). Custom Synthesis for Drug Discovery. Chemical & Engineering News, 81, 7, 75-78.

Santner, T. J., Williams B. J., and Notz, W. (2003). The Design and Analysis of Computer Experiments. New York, NY: Springer-Verlag Inc.

Venables, W. N., and Smith, D. M. (2002). An Introduction to R. Bristol, UK: Network Theory Ltd..

Vose, M. D. (1999). The Simple Genetic Algorithm: Foundations and Theory. Cambridge, Mass: MIT Press.

Wu, C. F. J., Mao, S. S. and Ma, F. S. (1990). SEL : A Search Method Based on Orthogonal Arrays. In S. Ghosh (1990), (Ed.), Statistical Design and Analysis of Industrial Experiments. New York, NY: Marcel Dekker, Inc., pp279 - 310.

Wu, C. F. J., and Hamada, M. (2000). Experiments: Planning, Analysis, and Parameter Design Optimization. New York, NY: Wiley.

APPENDIX A

INITIAL DESIGN AND FORBIDDEN ARRAY

A1. Initial Design

A	B	C	y
1	1	1	-2
1	1	37	-7
1	2	1	1
1	2	227	-7
1	3	1	1
1	3	227	-6
1	4	1	-3
1	6	6	5
1	6	14	-1
1	6	30	-2
1	6	32	7
1	6	54	-4
1	9	6	-4
1	10	30	-6
1	13	9	-8
1	13	10	-5
1	16	172	11
1	17	13	-6
1	22	3	-1
1	22	28	-4
1	22	46	1
1	23	8	5
1	27	8	1
1	27	34	-6
1	28	8	-7
1	28	20	1
2	2	6	2
2	2	30	-1
2	3	6	7
2	3	14	-4
2	3	30	3
2	5	6	3
2	9	9	-1
2	9	35	-7
2	9	172	0

2	12	13	-5
2	12	46	0
2	13	28	-1
2	13	46	9
2	14	13	1
2	14	40	7
2	14	46	0
2	15	13	1
2	15	40	11
2	15	46	4
2	24	1	0
2	24	227	-1
2	28	1	0
2	28	27	-2
2	28	227	-4
3	1	9	-1
3	1	10	8
3	1	22	5
3	1	35	7
3	1	172	28
3	2	35	1
3	2	172	-3
3	3	10	-7
3	3	22	0
3	3	35	-3
3	3	172	-4
3	4	9	1
3	4	35	1
3	4	172	18
3	5	9	-6
3	5	10	-1
3	6	3	2
3	6	13	1
3	6	28	0
3	6	40	0
3	6	46	-1
3	7	3	2
3	7	13	0
3	7	28	2
3	7	40	0
3	7	46	3
3	8	3	-8
3	8	13	7
3	8	28	-1
3	8	40	4
3	8	46	5

3	9	13	1
3	9	28	-1
3	9	46	11
3	10	3	-12
3	10	13	2
3	10	28	5
3	10	40	3
3	10	46	-4
3	12	8	1
3	12	20	3
3	12	34	0
3	13	8	-7
3	13	20	0
3	13	34	-8
3	14	8	1
3	14	20	0
3	15	8	-4
3	15	20	1
3	15	34	3
3	16	20	-1
3	16	34	-1
3	16	58	2
3	17	27	-5
3	22	1	-3
3	22	11	0
3	22	27	-2
3	22	37	-3
3	22	227	6
3	23	6	0
3	23	30	3
3	24	6	-2
3	24	30	-3
3	24	32	-7
3	25	30	3
3	27	30	1
3	28	6	8
3	28	14	3
3	28	30	1
4	1	13	0
4	1	28	-7
4	1	46	-2
4	2	28	-7
4	2	46	2
4	3	3	-3
4	3	13	0
4	3	28	-4

4	3	40	1
4	3	46	-2
4	4	28	1
4	4	46	3
4	5	13	-3
4	5	28	-4
4	5	46	-6
4	6	8	-2
4	6	20	0
4	6	58	3
4	7	8	1
4	7	20	-2
4	7	34	2
4	8	8	-5
4	8	20	-7
4	8	58	-5
4	9	8	-4
4	10	8	-5
4	10	20	-6
4	10	34	-6
4	12	1	-4
4	12	27	-1
4	12	227	0
4	13	1	-3
4	13	27	4
4	14	1	-6
4	14	11	3
4	14	27	-4
4	14	37	0
4	14	227	2
4	15	1	-4
4	15	11	1
4	15	27	-2
4	15	37	-2
4	15	227	3
4	16	11	0
4	22	6	9
4	22	14	3
4	22	30	-1
4	23	172	-2
4	24	9	-1
4	24	35	5
4	24	172	0
4	28	9	-1
4	28	35	-1
4	28	172	-1

A2. Forbidden Array

A	B	C
3	10	3
1	23	10
1	23	23
1	25	10
2	21	57
2	23	57
2	29	32
2	29	57
2	30	57
2	32	57
2	33	57
2	34	17
2	34	32
2	34	34
2	34	36
2	34	37
2	34	45
2	34	54
2	34	56
2	34	57
2	34	179
3	23	10
3	23	23
3	25	21
3	25	23
3	25	24
4	21	57
4	23	10
4	23	21
4	23	29
4	23	139
4	23	171
4	23	185
4	23	214
4	25	21
4	25	24
4	25	52
4	25	125
4	25	139
4	25	171
4	25	185

4	25	214
4	25	222
4	25	224
4	25	229
4	29	32
4	29	57
4	30	57
4	33	57
4	34	32
4	34	34
4	34	36
4	34	37
4	34	45
4	34	54
4	34	56
4	34	57
4	34	179
5	18	119
5	18	136
5	18	148
5	18	153
5	18	156
5	18	181
5	18	182
5	18	195
5	18	197
5	19	148
5	20	148
5	20	181
5	20	182

APPENDIX B

SELC SAS PROGRAM

```
*************************************************************************;
*  SELC: A macro to perform sequential elimination of level combinations. *;
*                                                                         *;
*  Inputs:                                                                *;
*    dfile:      name and path of the initial design. The order of the    *;
*                variables in the data set must be as x1, x2,...,xp,y.    *;
*                                                                         *;
*    ffile:      name and path of the forbidden array. This forbidden     *;
*                is obtained on the basis of prior knowledge.             *;
*                                                                         *;
*    numoff:     the required number of offspring.                        *;
*                                                                         *;
*    strength:   the strength of forbidden array. The foribidden array    *;
*                will be construct in process and based on y value.       *;
*                                                                         *;
*    order:      the order of forbidden array.                            *;
*                                                                         *;
*    numfact:    the number of factors in initial design.                 *;
*                                                                         *;
*    dir:        an argument of telling the searching goal.               *;
*                = 1 Search for Maximum                                   *;
*                = 0 Search for Minimum                                   *;
*                                                                         *;
*    wt:         an argument to create y-related values used in selecting *;
*                parents.                                                 *;
*                = 1 Percent weighting                                    *;
*                = 0 Squared percent weighting                            *;
*                                                                         *;
*    alpha:      level of significance. Generally, alpha is 0.05.         *;
*                                                                         *;
*       Li:      true levels for each factor (xi). This should be input   *;
*                seperately between %mend and %SELC clauses.              *;
*                                                                         *;
*  Outputs:                                                               *;
*    next_gen:   a data set consists of generated offspring.              *;
*                                                                         *;
*************************************************************************;

*GLM to check signIFicance of variables*;
*inputds = initial data SET that USEr provides;
*outputds = data set with signIFicant factor names and numbers;
%MACRO sigcheck(inputds=, outputds=);
   ODS OUTPUT GLM.ANOVA.y.ParameterEstimates=GLMout;
   ODS LISTING CLOSE;
   PROC GLM DATA=&inputds;
     MODEL y = x1-x&numfact;
   RUN;
```

```
   ODS LISTING;

   DATA &outputds;
     SET glmout;
     IF ((probt < &alpha) AND (probt > 0));
       IF parameter = "Intercept" THEN DELETE;
     KEEP parameter;
   RUN;
%MEND sigcheck;


*MACRO to select level to mutate to;
*inputds = initial data set that USEr provides;
*mutfac = factor to be mutated;
*sigds = output data set from sigcheck MACRO;
*        contains the signIFicant factors;
*mutds = data set that contains the new level of mutfac factor;
%MACRO mutprob(inputds=, mutfac=, sigds=, mutds=);
  *determine IF mutfac is a signIFicant factor;
  DATA mutfac;
    parameter = "&mutfac";
  RUN;

  PROC SORT DATA=&sigds;
    BY parameter;
  RUN;

  DATA sig;
    MERGE mutfac(in=a) &sigds(in=b);
      BY parameter;
      IF a AND b;
  RUN;

  PROC SQL NOPRINT;
    SELECT count(*)
      INTO : n
      FROM sig;
  QUIT;

  DATA _NULL_;
    SET mutfac;
    LENGTH fac 8;
    fac = substr(parameter,2);
    CALL SYMPUT('facnum',fac);
  RUN;

  %LET facnum = %cmpres(&facnum);

  *if n = 0, THEN all levels get equal probability;
  *if n = 1, THEN weighted probability;
  %IF &n = 0 %THEN %do;
    DATA &mutds;
      &mutfac=ceil(&&L&facnum*ranuni(-1));
    RUN;
  %END;
  %ELSE %DO;
    DATA prob1;
```

```
        DO i=1 TO &&L&facnum;
          x&facnum = i;
            prob1 = 1/&&L&facnum;
            OUTPUT;
        END;
        DROP i;
      RUN;

    PROC SQL NOPRINT;
        SELECT sum (y_pps)
        INTO : sum_y
        FROM &inputds;
      quit;

    PROC SORT DATA=&inputds;
        BY &mutfac;
      RUN;

    PROC MEANS DATA=&inputds SUM NOPRINT;
        BY &mutfac;
        VAR y_pps;
        OUTPUT OUT=outsum(drop=_TYPE_ _FREQ_) SUM=sum;
      RUN;

      DATA prob2;
        SET outsum;
        prob2 = sum/&sum_y;
      RUN;

      DATA merge_p;
        MERGE prob1 prob2;
        BY &mutfac;
        IF prob1 = . THEN prob1 = 0;
        IF prob2 = . THEN prob2 = 0;
        IF prob2 = 0 THEN prob = 0.25*prob1;
        IF prob2 > 0 THEN prob = 0.25*prob1 + 0.75*prob2;
      RUN;

    PROC SURVERSELECT data=merge_p method=pps sampsize=1
out=&mutds(keep=&mutfac) noprint;
      size prob;
    RUN;
  %END;
%MEND mutprob;

%MACRO SELC(ofile=, dfile=, ffile=, numoff=, strength=, order=, numfact=,
dir=, wt=, alpha=);

  DATA initial;     /* Data of initial design */
    INFILE &dfile EXPANDTABS;
    INPUT x1-x&numfact y;
  RUN;

  *Get data READy for follow-up PROCessing;
  PROC SQL NOPRINT;
    SELECT max(y)
        INTO : max_y
```

```
      FROM initial;

    SELECT min(y)
      INTO : min_y
      FROM initial;
QUIT;

*Create work and forbidden files;
%IF &dir=1 %THEN %do;
  DATA work_ds;
      SET initial;
      y_pps = (y - &min_y) / (&max_y - &min_y);
    RUN;

  PROC SORT DATA=initial OUT=forbid1(keep=x1-x&numfact);
    BY y;
    RUN;
%END;
%ELSE %IF &dir=0 %THEN %do;
  DATA work_ds;
      SET initial;
      y_pps = (&max_y - y) / (&max_y - &min_y);
  RUN;

  PROC SORT DATA=initial OUT=forbid1(keep=x1-x&numfact);
      BY DESCENDING y;
    RUN;
%END;

DATA work_ds;
  SET work_ds;
  %IF &wt=0 %THEN %do;
    y_pps = y_pps**2;
  %END;
RUN;

DATA forbid1;
  SET forbid1 (obs=&strength);
RUN;

DATA forbid2;
  INFILE &ffile EXPANDTABS;
    INPUT x1-x&numfact;
RUN;

*Check for signIFicance of factors;
%sigcheck(inputds=initial,outputds=sigfac);

DATA next_gen;
RUN;

*Generate new offspring;
%LET index = 0;
%DO %WHILE (&index < &numoff);

   *get mutation location;
   DATA _NULL_;
```

```
      loc=ceil(&numfact*ranuni(-1));
        CALL SYMPUT('ml',loc);
    RUN;
    %LET ml=%cmpres(&ml);   * ml is randomely selected mutation location;
    %LET mutfac=x&ml;       * mutfac represents mutation factor;

      *get mutation probabilities for mutation location;
    %mutprob(inputds=work_ds,mutfac=&mutfac,sigds=sigfac,mutds=mutval);

    ****************************************************************;
    *   Choose 2 parents with probability poportional to y values     ;
    ****************************************************************;
    PROC SURVEYSELECT data=work_ds method=pps sampsize=2
out=parents(keep=x1-x&numfact) NOPRINT;
      size y_pps;
    RUN;

    ************************************************;
    *    Complete steps                            ;
    *      1)Crossover, 2)Mutation, 3)Eligibility      ;
    ************************************************;
    PROC IML;
      USE parents;
      READ all into parents;
      n = &numfact;

        *crossover;
      c1 = int(&numfact*ranuni(-1));
      IF c1 = 0 THEN offsp = parents[1,1:n];
      ELSE offsp = parents[1,1:c1]||parents[2,(c1+1):n];

        *mutation;
      USE mutval;
        READ all into mutval;

        c2 = &ml;
        IF c2 = 1 THEN offsp = mutval[1,1]||offsp[1,2:n];
      IF c2 > 1 THEN IF c2 < n THEN offsp = offsp[1,1:(c2-
1)]||mutval[1,1]||offsp[1,(c2+1):n];
        ELSE offsp = offsp[1,1:(c2-1)]||mutval[1,1];
        varnames = 'x1':"x&numfact";
        CREATE offsp from offsp[colname=varnames];
        appEND from offsp;

        *eligibility;
        *make sure that the new offspring is not in the FA;
        *and is not in the original set or in the new generation;
      *offspring is prohibited if all elements of offspring is same as those
in forbid2;

        USE forbid2;
        READ all into forbid2;

        nf = nrow(forbid2);
        offsp_n = j(nf,1,1)*offsp;
        comp = (forbid2=offsp_n);
        max_same = max(comp[,+]);
```

```
        CREATE max_same from max_same[colname='max_same'];
      appEND from max_same;

      *offspring is prohibited if it has order-number same elements as those
in forbid1;

        USE forbid1;
        READ all into forbid1;

        offsp_s = j(&strength,1,1)*offsp;
        comps = (forbid1=offsp_s);
        max_sames = max(comps[,+]);
        CREATE max_sames from max_sames[colname='max_sames'];
      APPEND from max_sames;


        USE work_ds;
        READ all var('x1':"x&numfact") into work_ds;

        USE next_gen;
        READ all into next_gen;

        allgen = work_ds//next_gen;
      ng = nrow(allgen);
        offsp_n = j(ng,1,1)*offsp;
        comp = (allgen=offsp_n);
        same = max(comp[,+]);
        CREATE same from same[colname='same'];
      APPEND from same;

    QUIT;

  DATA _NULL_;
      SET max_same;
      CALL SYMPUT('max_same',max_same);
  RUN;

    DATA _NULL_;
      SET max_sames;
      CALL SYMPUT('max_sames',max_sames);
    RUN:

    DATA _NULL_;
      SET same;
      CALL SYMPUT('same',same);
    RUN;

  %IF &max_same < &numfact %THEN %DO;
     %IF &max_sames < &order %THEN %DO;
       %IF &same < &numfact %THEN %DO;
         %LET index = %EVAL(&index + 1);

         DATA next_gen;
           SET next_gen offsp;
             IF x1 = . THEN DELETE;
         RUN;
       %END;
```

```
      %END;
    %END;
  %END;

  DATA next_gen;
    SET next_gen;
  RUN;

  PROC SORT DATA=next_gen;
    BY x1-x&numfact;
  RUN;

  PROC print DATA=next_gen;
    TITLE1'SELC Algorithm';
      TITLE2'Output';
  RUN;

%MEND SELC;

*The USEr defines the maximum number of levels for each factor;
%LET  L1 = ;
  .
  .
  .
%GLOBAL L1 . . .;

*****************************************************;
*                                                   ;
* Here, USEr should specIFy:                        ;
*                                                   ;
*  1)      dfile:  dataset of initial design        ;
*  2)      ffile:  dataset of forbidden array       ;
*  3)     numoff:  # of required offspring          ;
*  4)   strength:  strength                         ;
*  5)      order:  order                            ;
*  6)    numfact:  total # of factors               ;
*  7)        dir:  =1 maximun wanted                ;
*                  =0 minimum wanted                ;
*  8)         wt:  =1 percent weighting             ;
*                  =0 squared percent weighting     ;
*  9)      alpha:  Cutoff for testing significance  ;
*                  Generally, it is 0.05            ;
*                                                   ;
*****************************************************;

OPTIONS mlogic mprint symbolgen;
%SELC(dfile=,
      ffile=,
      numoff=,
      strength=,
      order=,
      numfact=,
      dir=,
      wt=,
      alpha=);
```

APPENDIX C

SELC MATLAB PROGRAM

```
function [newruns,newforbiddenarray] =
SELC(strength,order,number_of_offspring,level_true)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%
%  This function implements the SELC algorithm.
%
%   [newruns,newforbiddenarray] =
%   SELC(strength,order,number_of_offspring,level_true)
%
%
%       strength = strength of the forbidden array, number of runs to be
%       added to the forbidden array based on the initial data
%       order = order of forbidden array. It is safe to put order = number
%       of factors.
%       number_of_offspring = total number of new offspring required
%       level_true = the possible levels of each factor, coded as 1,2,3,..
%
%       Example : Let there are four factors A, B, C and D with possible
%       levels 15, 5, 32, 109 respectively.
%       That is, A has levels 1, 2, 3, ..., 109
%                B has levels 1, 2, 3, 4, 5 etc
%       Hence level_true = [15 5 32 109];
%
%
%
%
%       Useage : [a,b] = SELC(3,4,10,[15 5 32 109])
%
%
%
%       forbidden array has to be given in one text file called
%       forbiddenarray.txt (no header). If there are three factors, A, B, C
%       then the text file should look like this
%
%        3  10  3
%        1  23  10
%        1  23  23
%        1  25  10
%        2  21  57
%        2  23  57
%
%
%
%       initial design  has to be given in one text file called
```

```
%       initialdesign.txt (no header).If there are three factors, A, B, C,
%       then the text file should look like this (first 3 columns are for
%       A, B and C; and the last column is for the response y
%
%        1  1   1   -2
%        1  1   37  -7
%        1  2   1   1
%        1  2   227 -7
%        1  3   1   1
%        1  3   227 -6
%        1  4   1   -3
%        1  6   6   5
%
%
%       Output : a = newruns : this will contain the list of new runs
%       Output : b = newforbiddenarray : this will contain the list of new
%       forbidden runs
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
% Here we first read the Initial Design and Forbidden array
%
%
% NOTE THAT THE LEVELS OF X HAVE TO BE POSITIVE INTEGERS (1,2,3,...)
%
forbiddenarray = textread();
initialdesign = textread();



%
%  m is # of columns of initialdesign
%  num_main_effects is the number of factors
%
m = size(initialdesign,2);
num_main_effects = m-1;

if size(level_true(:),1) ~= num_main_effects
    disp('Check the number of factors');
    newruns = [];
    newforbiddenarray = [];
    return;
end

%
% we create matries x, y represent X (compounds) and Y (response)
% respectively here we substract the mean of y
%
x = initialdesign(:,1:num_main_effects);
y = initialdesign(:,m);
yc = y-mean(y); %y has been centered to zero, so no intercept in fitting the
regression line

for i = 1:num_main_effects
    if (length(unique(x(:,i))) > level_true(i))
```

```
    disp(sprintf('Check the levels of factor %i.' ,i))
    newruns = [];
    newforbiddenarray = [];
    return;
    end
end

%Identify the significant factors
sigfactor = evalsig(yc,x);

newruns = [];

while size(newruns,1) < number_of_offspring

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %                                                           %
    %         Now we do the reproduction                        %
    %                                                           %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %
    % We choose 2 "good candidates" to reproduce.
    %
    population_size = 2;
    candidates = choose_with_prob(y,population_size);

    p1 = x(candidates(1),:);
    p2 = x(candidates(2),:);

    % Now we let these two parents, namely p1 and p2, reproduce
    % and create new generation p1new and p2new

         %
         % First do the crossover
         %
         [p1new p2new] = crossover(p1,p2);

         %
         % Now do mutation
         %
         p1new = mutation(x,y,sigfactor,p1new,num_main_effects,level_true);
         p2new = mutation(x,y,sigfactor,p2new,num_main_effects,level_true);


    % Now we add new runs to the forbidden array based on the Initial design

    yy = unique(sort(y));
    index = 0.*[1:size(y)]';
    ijk = 1;
    while sum(index)<strength
        index = index + (y==yy(ijk));
        ijk = ijk+1;
    end
    index = index.*[1:size(y)]';
```

```
    index = index(index>0);
    newforbiddenarray = x(index,:);


    % Check whether these two level combanations are allowed or not
    % if indicator1 is equal to 1, then p1new is allowed
    % if indicator2 is equal to 1, then p2new is allowed
    indicator1 = check(newforbiddenarray,p1new,order) &
check(forbiddenarray,p1new,num_main_effects );
    indicator2 = check(newforbiddenarray,p2new,order) &
check(forbiddenarray,p2new,num_main_effects );

    % Check whether these two level combanations are new or not
    % if indicator3 is equal to 1, then p1new is new
    % if indicator4 is equal to 1, then p1new is new
    if indicator1 == 1
        indicator3 = check(x,p1new,num_main_effects);
        if (indicator3 == 1)
            % If the run is not forbidden and is a new one, save it
            newruns = [newruns; p1new];
        end
    end
    if indicator2 == 1
        indicator4 = check(x,p2new,num_main_effects);
        if (indicator4 == 1)
            % If the run is not forbidden and is a new one, save it
            newruns = [newruns; p2new];
        end
    end


end


newruns = newruns(1:number_of_offspring,:);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function sigfactor = evalsig(y,x);

X = modelmatrix(x);
X = [ones(size(X,1),1) X];
[B,BINT,R,RINT,STATS] = regress(y,X,0.05);
sigfactor = [];

for i = 2:size(BINT,1)
    sigfactor(i-1) = (BINT(i,1) > 0) | (BINT(i,2) < 0);
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function X = modelmatrix(D)
% given model matrix D, this function generates the design matrix X
% NOTE : We do _NOT_ have the intercept term here
% here we assume all factors are at 11 level
% Create Complete Model Matrix - with main effects, quadratic effects and
2fi's (linear by linear only).
% The main effects A,B,C,D,...
% Then comes the quadratic effects A^2, B^2, C^2, D^2 ...
% Then comes the interactions Yates' order AB, AC, AD, ...
%
%  We make a change in coading. We use linear quadratic system
%
k = length(D(1,:));
%
X = [];
% First the main effects - standardized to have norm 1
for i = 1:k
    X = [X D(:,i)/norm(D(:,i),2)];
end
% Then the quadratic effects
X = [X X.^2];
% Then the interaction effects
l = 2*k + 1;
for i = 1:k
    for j = (i+1):k
        X(:,l)=X(:,i).*X(:,j);
        l = l + 1;
    end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function candidates = choose_with_prob(x,number)
%
% This function chooses 'number'-many _distinct_ elements of x with
probability
% proportional  to the value of x
%
% candidates = indices
x = x(:);
x =  x.*(x > 0);

prob = x/sum(x);
cumulative_prob = cumsum(prob);

candidates = [];

% change the state
%rand('state',sum(10^10*clock));
```

```
while (size(candidates) < number)
    u = rand(1);
    temp =   (u >= cumulative_prob) .* (1:length(prob))' ;
    candidates = [candidates max(temp)+1];
    candidates = unique(candidates);
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [p1new, p2new] = crossover(p1,p2)
% this function does the crossover

% change the state
%rand('state',sum(10^10*clock));

% first choose the crossover location
crossover_location = ceil(rand(1)*length(p1));

% now do the crossover
p1new = [ p1(1:crossover_location-1) p2(crossover_location:length(p1))];
p2new = [ p2(1:crossover_location-1) p1(crossover_location:length(p1))];


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function p1 = mutation(x,y,sigfactor,p1,num_main_effects,level_true)
% this function does the crossover

% change the state
%rand('state',sum(10^10*clock));

% first choose the mutation location
mutation_location = ceil(rand(1)*length(p1));


% Check whether this location is important or not
if sigfactor(mutation_location) == 1

   i = identify(mutation_location,sigfactor,num_main_effects);
   % If this is only for main effect or quadratic effect, then the mutation
   % will be on that effect only. Otherwise, it will be an interaction
   % effect,  and the mutation will be joint on both of the parent factors.

   if length(i)==1
       % this is only for main effect
       p1 = one_factor_mutation(p1,x,y,i,level_true);
   else
       % this is for quadratic effect
```

```
        p1 = two_factor_mutation(p1,x,y,i(1),i(2),level_true);
    end

else
% if the factor is not imp, then just do the mutation
    p1(mutation_location) = ceil(rand(1)*level_true(mutation_location));
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function interaction_number = is_there_imp_2fi(k,sigfactor,num_main_effects)

% if interactiton_number = 0 then no interaction is imp. otherwise this is
% the number corresponding to _one_of_the_ imp interactions

% The nonzero entries of the (k-1)th column and (k)th row of the matrix
% facmat corresponds to the 2fi's of factor k

% example : Let there be 4 factors A, B, C and D and AC interaction is
% significant. Then for k = 1 and 3 (i.e. for A and C), it will return 10
% (i.e. AC) and for k = 2 and 4, it will return 0 (i.e. No important
% interactions)
%

sigfactor = sigfactor(:);
facmat = zeros(num_main_effects-1,num_main_effects-1);
count = 1;
for ( i = 1:num_main_effects-1)
    for (j = i:num_main_effects-1)
        facmat(i,j) = count;
        count = count + 1;
    end
end
if (k == 1)
    values = facmat(1,:);
elseif (k==num_main_effects)
    values = facmat(:,k-1);
else
    values = [facmat(:,k-1)' facmat(k,:)]';
end
values = values(values>0) + 2*num_main_effects;
values = values(:);
interaction_number = max(values.*sigfactor(values));


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function z = identify(i,sigfactor,num_main_effects)
%
% Given a mutation location, this function identifies the parent factors.
```

```
% For example , if there are 4 factors, A, B, C, D and
% if A and AD are important, then
% i = 1 will return A and D (D will come as AD interaction is significant)
% i = 2 will return B
% i = 3 will return C
% i = 4 will return A and D (A will come as AD interaction is significant)
%

% i = effect number
% z = main effect/quadratic effect/parents
value = i; % this is the indicator of presence of significant 2fi.

    if ( i <= num_main_effects)
        % then it is a main effect
        index = i;
        %check whether it has any imp two factor interaction
        value = is_there_imp_2fi(index,sigfactor,num_main_effects);

    elseif ( i <= 2*num_main_effects)
        % then it is a quadratic effect
        % index is the corresponding main effect
        index = i-num_main_effects;
        value = is_there_imp_2fi(i,sigfactor,num_main_effects);

    end

    if (value == 0)
        z = index;
    else
        % it is a linear by linear interaction effect

        % must first decide which effects are parents
        % the interaction is the int_num^th interaction from Yates' order
        %
        % count is now the number of the first parent
        % count1 to be computed is the number of the second parent

        %For this part, all credit goes to Derek Bingham
        i = value;

        int_num=i-2*num_main_effects;

        count=0;
        count1=int_num;
        num_ints=0;
        while count1>0
            count=count+1;
            count1=count1-(num_main_effects-count);
        end
        count1=0;
        for j=1:(count-1)
            count1=count1+(num_main_effects-j);
        end
        count1=int_num-count1;
        count1=count+count1;
```

```
        z = [count count1];
    end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function p1 = one_factor_mutation(p1,x,y,i,level_true)
%
% This function calculates mutation for factor i with weight scheme given
% by the vector prob.
%

% first calculate the probability
prob1 = prob_main_effect(x,y,i,level_true);

% then do the actual mutation
cumulative_prob1 = cumsum(prob1);
u = rand(1);
temp =   (u >= cumulative_prob1) .* (1:length(prob1))' ;
p1(i) = max(temp);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function prob = prob_main_effect(x,y,i,level_true)

%
% Here we calculate the mutation probability of the ith factor for
% different levels
%

% First we identify the possible and observed levels. Note that in the
initial array we
% do not have all levels of each factors.

num_level_true = level_true(i);
level_observed = unique(x(:,i));
num_level_observed = size(level_observed,1);

% Now we calculate the mean of the y-values corresponding to each observed
% levels.


% first we create a column of zeros for corresponding to the possible
% levels of the factor
ymean = zeros(num_level_true,1);
for j = 1:num_level_observed
     ymean(level_observed(j)) = mean(y(x(:,i)==level_observed(j)));
end
```

```
% Now we consider only positive values of this mean vector
ymean = ymean .* (ymean > 0);
ymean = ymean/sum(ymean);

% uniform prob vector
uniform_prob = 1/num_level_true * ones(num_level_true,1);


% Here we calculate the final probability
alpha = (num_level_observed / num_level_true) ;
prob = alpha*ymean + (1-alpha)*uniform_prob;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function p1 = two_factor_mutation(p1,x,y,i,j,level_true)
%
% This function calculates mutation for factors i and j with weight scheme
given
% by the vector prob.
%

% first calculate the probability
prob = prob_2fi(x,y,i,j,level_true);
cumulative_prob = cumsum(prob(:));
u = rand(1);
temp =   (u < cumulative_prob) .* ([1:length(prob(:))]') ;
index = min(temp(temp>0));
number_of_rows = level_true(i);
number_of_col = level_true(j);

row = index - (ceil(index/number_of_rows)-1)*number_of_rows;
col = ceil(index/number_of_rows);

p1(i) = row;
p1(j) = col;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function prob = prob_2fi(x,y,i,j,level_true)
%
% Here we calculate the mutation probability of the ith and jth factor for
% different levels
%
yprob = zeros(level_true(i),level_true(j));

% Now we calculate the mean of the y-values corresponding to each observed
% levels.
```

```
for ik = 1:level_true(i)
    for jk = 1:level_true(j)
        if size(y((x(:,i)==ik) & (x(:,j)==jk)),1) > 0
            yprob(ik,jk) = mean(y((x(:,i)==ik) & (x(:,j)==jk)));
        end
    end
end

yprob = yprob.*(yprob>0);
yprob = yprob/sum(sum(yprob));


% uniform prob vector
uniform_prob = 1/(level_true(i)*level_true(j)) *
ones(level_true(i),level_true(j));


alpha = sum(sum(yprob > 0))/(level_true(i)*level_true(j));
prob = alpha.*yprob + (1-alpha).*uniform_prob ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function check = check(oldmatrix,newrow,order)
%
% This function compares the elements of the rows of a matrix (oldmatrix)
% with that of a new row (newrow). If the 'newrow' and any row of
% 'oldmatrix' have at least 'order'-many elements common (at the same
% locations), then this function will return 0, 1 otherwise.
%
% example : oldmatrix = [ 1 1 0 1 ;
%                         2 0 1 1];
%           newrow = [ 1 2 0 2];
%           order = 2;
%
% then check = 0
n = size(oldmatrix,1);
check = 1;
for i = 1:n
    if ( sum(newrow == oldmatrix(i,:)) >= order )
        check = 0;
        break;
    end
end
```

APPENDIX D

SELC R PROGRAM

```
#########################################################################
#
#   SELC
#     This R program of SELC is composed of 12 modules(functions) which
#   are:
#
#   1) evalsig / modelmatrix
#   2) choose_with_prob
#   3) crossover
#   4) is_there_imp_2fi
#   5) identify
#   6) prob_main_effect
#   7) one_factor_mutation
#   8) prob_2fi
#   9) two_factor_mutation
#  10) mutation
#  11) check
#  12) SELC
#
#
#   In order to complete the program, user should specify 4 variable:
#
#   1) strength
#   2) order
#   3) level_true
#   4) number_of_offspring
#
#########################################################################


#########################################################
#
#    Here, user should specify 4 variables:
#    1) strength
#    2) order
#    3) level_true
#    4) number_of_offspring
#
#########################################################
strength=
order=
level_true=
number_of_offspring=


#########################################################
#
# Read basic information and create some variables
```

```
# would be used in most functions
#
#####################################################

initialdesign=read.table("")
forbiddenarray=read.table("")
initialdesign=as.matrix(initialdesign)
forbiddenarray=as.matrix(forbiddenarray)

# Here we first read the Initial Design and Forbidden array
# NOTE THAT THE LEVELS OF X HAVE TO BE POSITIVE INTEGERS (1,2,3,...)

#
#  Now we input the other variables  specified by the user
#  numoff is # of offspring desired
#  m is # of columns of initialdesign
#
m = ncol(initialdesign)
num_main_effects = m-1

#
# we create matries x, y represent X (compounds) and Y (response)
respectively
# here we substract the mean of y
#
x = initialdesign[,-m]
x = as.matrix(x)
y = initialdesign[,m]
y = as.matrix(y)
yc = y-mean(y)
#yc has been centered to zero, so no intercept in fitting the regression line

#########################################################################
#
# This function helps construct design matrix
#
#########################################################################

modelmatrix=function(D)
{
# given model matrix D, this function generates the design matrix X
# NOTE : We do _NOT_ have the intercept term here
# here we assume all factors are at 11 level
# Create Complete Model Matrix - with main effects, quadratic effects and
2fi's (linear by
# linear only).
# The main effects A,B,C,D,...
# Then comes the quadratic effects A^2, B^2, C^2, D^2 ...
# Then comes the interactions Yates' order AB, AC, AD, ...
#
#  We make a change in coading. We use linear quadratic system

k = ncol(D)
l = nrow(D)

X = c()
```

```
# First the main effects - standardized to have norm 1
p=2

for (i in 1:k)
  {
   X = cbind(X,D[,i]/(sum(abs(D[,i])^p)^(1/p)))
  }

# Then the quadratic effects

X = cbind(X,X^2)

# Then the interaction effects
c = 1
f = factorial(k)/(2*factorial(k-2))
Y = matrix(0,l,f)

for (i in 1:(k-1))
  {
   for (j in (i+1):k)
      {
        Y[,c]=X[,i]*X[,j]
        c=c+1
      }
   }
X=cbind(X,Y)

return(X)
}

#############################################################################
#
#  The Functions are for checking the importance of each factor
#
#############################################################################


evalsig=function(x,y)
{
x = modelmatrix(x)
g = lm(y ~ x)
sg = summary(g)
sgcoef = sg$coef
sgcoef = as.matrix(sgcoef)
pvalues = sgcoef[,4]

n = length(pvalues)

sigfactor=matrix(0,1,n)

for (i in 1:n)
  {
   if (pvalues[i]<0.05) sigfactor[i] = 1
  }

return(sigfactor)
}
```

```
##########################################################################
#
# This function chooses 'number'-many _distinct_ elements of x with
probability
# proportional  to the value of x
#
##########################################################################

choose_with_prob=function(x,number)
{
# candidates = indices
k = ncol(x)
l = nrow(x)
x = matrix(x,k*l,1)
x = x*(x > 0)

prob = x/sum(x)
cumulative_prob = cumsum(prob)

candidates = c()

# change the state
#rand('state',sum(10^10*clock))

while (NROW(candidates) < number)
  {
    u = runif(1)
    temp =   (u >= cumulative_prob)* matrix(1:nrow(prob))
    candidates = rbind(candidates, max(temp)+1)
    candidates = unique(candidates)
  }

return(candidates)
}


#######################################################
#
# This function does the crossover
#
#######################################################

crossover=function(p1,p2)
{

# first choose the crossover location
crossover_location = floor(runif(1)*length(p1)+1)

# now do the crossover
p1new = c( p1[1:crossover_location-1], p2[crossover_location:NROW(p1)])
p2new = c( p2[1:crossover_location-1], p1[crossover_location:NROW(p1)])

newoff = rbind(p1new, p2new)
```

```
return(newoff)
}



################################################
#
# This function helps check whether there is
# significant interaction
#
################################################

is_there_imp_2fi=function(k,sigfactor,num_main_effects)
{

# if interactiton_number = 0 then no interaction is imp. otherwise this is
# the number corresponding to _one_of_the_ imp interactions

# The nonzero entries of the (k-1)th column and (k)th row of the matrix
# facmat corresponds to the 2fi's of factor k

# example : Let there be 4 factors A, B, C and D and AC interaction is
# significant. Then for k = 1 and 3 (i.e. for A and C), it will return 10
# (i.e. AC) and for k = 2 and 4, it will return 0 (i.e. No important
# interactions)
#

n=length(sigfactor)
sigfactor = matrix(sigfactor,n,1)
facmat = matrix(0,num_main_effects-1,num_main_effects-1)
count = 1

for ( i in 1:(num_main_effects-1))
 {
  for ( j in i:(num_main_effects-1))
     {
        facmat[i,j] = count
        count = count + 1
     }
  }

if (k == 1) {values = facmat[1,]} else if (k==num_main_effects) {values =
facmat[,k-1]} else {values = c(facmat[,k-1], facmat[k,])}

values = values[values>0] + 2*num_main_effects
r=length(values)
values = matrix(values,r,1)
interaction_number = max(values*sigfactor[values])

return(interaction_number)
}



#############################################################################
#
# Given a mutation location, this function identifies the parent factors.
```

```
# For example , if there are 4 factors, A, B, C, D and
# if A and AD are important, then
# i = 1 will return A and D (D will come as AD interaction is significant)
# i = 2 will return B
# i = 3 will return C
# i = 4 will return A and D (A will come as AD interaction is significant)
#
#########################################################################

identify=function(i,sigfactor,num_main_effects)
{
# i = effect number
# z = main effect/quadratic effect/parents

value = i # this is the indicator of presence of significant 2fi.

    if ( i <= num_main_effects)
       {
        index = i
        # then it is a main effect
        #check whether it has any imp two factor interaction
        value = is_there_imp_2fi(index,sigfactor,num_main_effects)
       } else if ( i <= 2*num_main_effects)
       {
        # then it is a quadratic effect
        # index is the corresponding main effect
        index = i-num_main_effects
        value = is_there_imp_2fi(i,sigfactor,num_main_effects)
        }


    if (value == 0)
        {
         z=index
        } else
       {
        # it is a linear by linear interaction effect

        # must first decide which effects are parents
        # the interaction is the int_num^th interaction from Yates' order
        #
        # count is now the number of the first parent
        # count1 to be computed is the number of the second parent

        #For this part, all credit goes to Derek Bingham
        i = value

        int_num=i-2*num_main_effects

        count=0
        count1=int_num
        num_ints=0
        while (count1>0)
           {
            count=count+1
            count1=count1-(num_main_effects-count)
           }
```

```
        count1=0
        for (j in 1:count-1 )
          {
           count1=count1+(num_main_effects-j)
          }
        count1=int_num-count1
        count1=count+count1
        z = cbind(count, count1)
       }

return(i)
}



#########################################################################
#
# Here we calculate the mutation probability of the ith factor for
# different levels
#
#########################################################################

prob_main_effect=function(x,y,i,level_true)
{
# First we identify the possible and observed levels. Note that in the
initial array we
# do not have all levels of each factors

num_level_true = level_true[i]
level_observed = sort(unique(x[,i]))
num_level_observed =length(level_observed)

# Now we calculate the mean of the y-values corresponding to each observed
# levels


# first we create a column of zeros for corresponding to the possible
# levels of the factor
ymean = c(1:num_level_true*0)
for (j in 1:num_level_observed)
{
 ymean[level_observed[j]] =
mean(initialdesign[,m][initialdesign[,i]==level_observed[j]])
}

# Now we consider only positive values of this mean vector
ymean = ymean * (ymean > 0)
ymean = ymean/sum(ymean)

# uniform prob vector
uniform_prob = 1/num_level_true * c(1:num_level_true*0+1)

# Here we calculate the final probability
alpha = (num_level_observed / num_level_true)
prob = alpha*ymean + (1-alpha)*uniform_prob

return(prob)
```

```
}


#####################################################################
#
# This function calculates mutation for factor i with weight
# scheme given by the vector prob
#
#####################################################################

one_factor_mutation=function(p1,x,y,i,level_true)
{

# first calculate the probability
prob1 = prob_main_effect(x,y,i,level_true)

# then do the actual mutation
cumulative_prob1 = cumsum(prob1)
u = runif(1)
temp =   (u >= cumulative_prob1) * (1:length(prob1))
p1[i] = max(temp)

return (p1)
}



#########################################################################
#
# Here we calculate the mutation probability of the ith and jth factor for
# different levels
#
#########################################################################

prob_2fi=function(x,y,i,j,level_true)
{

yprob = matrix(0,level_true[i],level_true[j])

# Now we calculate the mean of the y-values corresponding to each observed
# levels.

for (ik in 1:level_true[i])
  {
    for (jk in 1:level_true[j])
     {
      if ((length(initialdesign[,m][initialdesign[,i]==ik &
initialdesign[,j]==jk])) > 0) {
      yprob[ik,jk] = mean(initialdesign[,m][initialdesign[,i]==ik &
initialdesign[,j]==jk])
      }
     }
  }
yprob = yprob*(yprob>0)
yprob = yprob/sum(sum(yprob))
```

```
# uniform prob vector
uniform_prob = 1/(level_true[i]*level_true[j]) *
matrix(1,level_true[i],level_true[j])

alpha = sum(sum(yprob > 0))/(level_true[i]*level_true[j])
prob = alpha*yprob + (1-alpha)*uniform_prob

return(prob)
}



##############################################################################
#
# This function calculates mutation for factors i and j with weight scheme
given
# by the vector prob.
#
##############################################################################

two_factor_mutation=function(p1,x,y,i,j)
{
# first calculate the probability
prob = prob_2fi(x,y,i,j,level_true)
cumulative_prob = cumsum(prob)

u = runif(1)
temp =   (u < cumulative_prob) * (1:length(prob))
index = min(temp[temp>0])
number_of_rows = level_true[i]
number_of_col = level_true[j]

row = index - (floor(index/number_of_rows))*number_of_rows
col = floor(index/number_of_rows+1)

p1[i] = row
p1[j] = col

return(p1)
}



##########################################
#
#  this function does the mutation
#
##########################################

mutation=function(x,y,sigfactor,p1,num_main_effects,level_true)
{

# change the state
# rand('state',sum(10^10*clock))

# first choose the mutation location
```

```
mutation_location = floor(runif(1)*length(p1)+1)

# Check whether this location is important or not
 if (sigfactor[mutation_location] == 1)
 {
  {
   i = identify(mutation_location,sigfactor,num_main_effects)
   # If this is only for main effect or quadratic effect, then the mutation
   # will be on that effect only. Otherwise, it will be an interaction
   # effect,  and the mutation will be joint on both of the parent factors
   # if the factor is not imp, then just do the mutation

   if (length(i)==1){
                     p1 = one_factor_mutation(p1,x,y,i,level_true)
                    } else {
                              p1 =
two_factor_mutation(p1,x,y,i[1],i[2],level_true)
                             }
   }
  } else { # this is for main effect and for quadratic effect
         p1[mutation_location] =
floor(runif(1)*level_true[mutation_location]+1)
       }

return (p1)
}



#############################################################################
#
# This function compares the elements of the rows of a matrix (oldmatrix)
# with that of a new row (newrow). If the 'newrow' and any row of
# 'oldmatrix' have at least 'order'-many elements common (at the same
# locations), then this function will return 0, 1 otherwise.
#
# example : oldmatrix = [ 1 1 0 1
#                         2 0 1 1]
#           newrow = [ 1 2 0 2]
#           order = 2
#
# then check = 0
#
#############################################################################

check = function(oldmatrix,newrow,order)
{
oldmatrix=as.matrix(oldmatrix)
newrow=as.matrix(newrow)
n = NROW(oldmatrix)
check = 1
for (i in 1:n)
   {
    if ( sum(newrow == oldmatrix[i,]) >= order ) check = 0
    break
    }
```

```
return (check)
}



#########################################################################
#
#        This function implements the SELC algorithm.
#       strength = strength of the forbidden array, number of runs to be
#       added to the forbidden array based on the initial data
#       order = order of forbidden array. It is safe to put order = number
#       of factors.
#       number_of_offspring = total number of new offspring required
#       level_true = the possible levels of each factor, coded as 1,2,3,...
#
#       Example : Let there are four factors A, B, C and D with possible
#       levels 15, 5, 32, 109 respectively.
#       That is, A has levels 1, 2, 3, ..., 109
#                B has levels 1, 2, 3, 4, 5 etc
#       Hence level_true = [15 5 32 109]
#
#       Useage : [a,b] = SELC(3,4,10,[15 5 32 109])
#
#       Output : a = newruns : this will contain the list of new runs
#       Output : b = newforbiddenarray : this will contain the list of new
#                    forbidden runs
#
#########################################################################

SELC=function(strength, order, level_true, number_of_offspring)
{

if (length(level_true)!=num_main_effects)
  {
   print("Check the number of factors")
   newrums=matrix()
   newforbiddenarray=matrix()
   return
  }

for (i in 1:num_main_effects)
  {
    if (length(unique(x[,i]))>level_true[i])
    {
    desp(sprintf('Check the number of factors %.',i))
    newruns=matrix()
    neworbiddenarray=matrix()
    return
    }
   }

#Identify the significant factors
sigfactor = evalsig(x,yc)

newruns = c()
```

```
while (length(newruns) < (m-1)*number_of_offspring){

    ##############################################
    #
    #         Now we do the reproduction
    #
    ##############################################

    #
    # We choose 2 "good candidates" to reproduce.
    #
    population_size = 2
    candidates = choose_with_prob(y,population_size)

    p1 = x[candidates[1],]
    p2 = x[candidates[2],]

    # Now we let these two parents, namely p1 and p2, reproduce
    # and create new generation p1new and p2new

         #
         # First do the crossover
         #
         newoff = crossover(p1,p2)
         p1new = newoff[1,]
         p2new = newoff[2,]


         #
         # Now do mutation
         #
         p1new = mutation(x,y,sigfactor,p1new,num_main_effects,level_true)
         p2new = mutation(x,y,sigfactor,p2new,num_main_effects,level_true)


    # Now we add new runs to the forbidden array based on the Initial design

    yy = unique(sort(y))
    index = matrix(0*1:length(y))
    ijk = 1
    while ( sum(index) < strength)
        {
        index = index + (y==yy[ijk])
        ijk = ijk+1
        }

    index = index*matrix(1*1:length(y))
    index = index[index>0]

    newforbiddenarray = x[index,]



    # Check whether these two level combanations are allowed or not
    # if indicator1 is equal to 1, then p1new is allowed
    # if indicator2 is equal to 1, then p2new is allowed
```

```
    indicator11 = check(newforbiddenarray,p1new,order)
    indicator12 = check(forbiddenarray,p1new,num_main_effects )
    indicator21 = check(newforbiddenarray,p2new,order)
    indicator22 = check(forbiddenarray,p2new,num_main_effects )

    if ((indicator11+indicator12)==2) {indicator1=1} else {indicator1=0}
    if ((indicator21+indicator22)==2) {indicator2=1} else {indicator2=0}

    # Check whether these two level combanations are new or not
    # if indicator3 is equal to 1, then p1new is new
    # if indicator4 is equal to 1, then p1new is new

    if (indicator1 == 1)
       {
        indicator3 = check(x,p1new,num_main_effects)
        if (indicator3 == 1) {
                              newruns = rbind(newruns, p1new)
                             }
       }
            # If the run is not forbidden and is a new one, save it


    if (indicator2 == 1)
       {
        indicator4 = check(x,p2new,num_main_effects)
        if (indicator4 == 1)  {
                               newruns = rbind(newruns, p2new)
                              }
       }
            # If the run is not forbidden and is a new one, save it

}

return(newruns)
}
```