GOSET

Genetic Optimization System Engineering Tool

For Use with MATLAB®

Version 2.6
January 1, 2014

Coordinator: S.D. Sudhoff
(sudhoff@purdue.edu)

Contents

Forward

GOSET is a general purpose genetic algorithm package to support single- and multi-objective optimization. It has been used at extensively at a number of universities, companies, and laboratories for design optimization, particularly in power engineering and power electronic applications.

The GOSET software package is a direct result of three research awards from the Office of Naval Research. The software itself had its beginnings prior to any formal research support, at that time it was known as ENEGAT (ESAC Non-Encoded Genetic Algorithm Toolbox). Later, a substantial revision and expansion of the software were made under Office of Naval Research (ONR) support through the effort “Polytopic Model Based Stability Analysis and Genetic Design of Electric Warship Power Systems,” contract N00014-02-1-0990. In this regard I sincerely appreciate the support of both Katherine Drew, our program manager at ONR, and Ed Zivi, a Professor at the US Naval Academy, whose support of the vision of this effort was critical. Contractually, another important source of this effort was the effort “National Naval Responsibility for Naval Engineering: Education and Research for the Electric Naval Engineer,” contract N00014-02-1-0623. This award paid for the development of this manual, as well as a short course to go with it. I would like to express our appreciation to Terry Erickson for supporting the pedagogical and technology transfer effort. At this point of time GOSET is supported through the ONR’s Electric Ship Research and Development Consortium (N00014-08-1-0080). We appreciate ONR’s support in general and our current program manager’s support, Sharon Beerman-Curtin, in particular.

I also thank all of the students who helped me write the GOSET package. Brandon and Brant Cassimere, Chunki Kwon, Jim Cale, and Brian Kuhn all served as beta testers in the use of GOSET as it was developed. Dionysius Aliprantis played a key role for getting me interested in genetic algorithms in the first place. Stan Żak, a fellow faculty member and close collogue, helped sustain my interest and our research group with his scholarship and enthusiasm. Benjamin Loop contributed routines relating to the identification of non-dominated solutions as well as simulated binary crossover algorithms. Finally, I would especially like to express my thanks to Yonggon Lee, a former Phd student, who is responsible for putting together the first version of this manual.

Scott D. Sudhoff
Professor of Electrical and
Computer Engineering
Purdue University

Chapter 1

Introduction

1.1 The Genetic Optimization System Engineering Tool (GOSET)

The Genetic Optimization System Engineering Tool (GOSET) is a MATLAB® based code for solving optimization problems. In the course of its development, it was extensively used to solve a variety of engineering problems – particularly those related to magnetics, electric machinery, power electronics, and entire power and propulsion systems. It has been used to automatically design inductors, brushless dc motors, power supplies, and inverters and for the parameter identification of synchronous machines, induction machines, gas turbines, etc. It is meant primarily as an engineering tool, although it is quite generic in its ability to solve both single-objective and multi-objective optimization problems. Because it solves these problems using evolutionary algorithms it is very robust in its ability to seek global rather than local optimum, as well as in its ability to contend with functions that are not 'friendly' in that they are, for example, discontinuous. GOSET provides the means for the user to either be blissfully unaware of the algorithms and parameters used, or to become intimately involved in the exact algorithms as well as the parameters used in these algorithms. In short, it provides the user with a powerful tool for the automation of the engineering design process.

1.2 System Requirements

GOSET runs on MATLAB, and you can refer to the system requirements for corresponding versions of MATLAB. It does not require the use of any other MATLAB toolboxes. However, the MATLAB Paralleling Computing Toolbox will enable GOSET to utilize parallel computing.

1.3 Installing GOSET

GOSET is MATLAB based toolbox and the installation is a simple process of adding the GOSET path to the MATLAB paths. For users unfamiliar with how to do this, simply type 'help path' at the MATLAB prompt. The 'GOSET Core' is directory which should be added to the path.

1.4 New Features

GOSET 2.6 has one major new feature. For users who have MATLAB Parallel Computing Toolbox, GOSET 2.6 supports parallel processing. This is discussed in Section 5.11 under Fitness Evaluation.. For problems with computationally involved fitness functions, this can yield a many fold increase in computation speed.

Chapter 2

An Overview of Single-Objective Genetic Algorithms

This section is devoted to a brief overview of Genetic Algorithms (GAs) focused on the canonical genetic algorithm. A more extensive review, and one which is closely related to GOSET, is set forth in the book *Power Magnetic Devices: A Multi-Objective Design Approach* by S.D. Sudhoff and published by IEEE Press/Wiley.

2.1 Introduction to genetic algorithms

Genetic algorithms are optimization methods that are inspired by biological evolution. GAs operate on a population of candidate solutions and apply the principle of survival of the fittest to evolve the candidate solutions towards the desired optimal solutions.

In GAs, candidate solutions are referred to as *individuals*. The defining properties of these individuals (parameters) are encoded to chromosomes that consist of a string of genes. According to the representation rule, a gene can be a symbol from an alphabet (in a canonical GA), a binary number, integer, real-value, etc. A *population* refers to the group of individuals.

The *fitness* of an individual is a metric that tells us how good each individual is as the solution to the given problem. Using a fitness function, individuals are assigned corresponding fitness values. The individuals with better fitness values are more like to survive and reproduce.

With the representation rule and the fitness function determined for the given optimization problem, an initial population is randomly generated and fitness values are evaluated. Then a pair of *parent* chromosomes is selected from the current population. The probability of selection increases with increasing fitness. Genetic operators such as *crossover* and *mutation* are applied to these parent chromosomes to generate children. The children are used to create a new population, for which fitness values are evaluated and assigned. This process of selection, crossover, mutation, and fitness evaluation is repeated until a stopping criterion is satisfied. Each iteration of this procedure is called a generation.

From the above description of a GA, it is clear that GAs are radically different from the classical optimization approaches. Some of the most significant differences are:

- GAs operate encodings of the parameter values, not necessarily the actual parameter values
- GAs operate on a population of solutions, not a single solution
- GAs only uses the fitness values based on the objective functions and do not require derivative information or other knowledge
- GAs uses probabilistic computations, not deterministic ones
- GAs are efficient in handling problems with a discrete search space

2.2 Canonical genetic algorithm

In this section, a canonical GA is introduced to illustrate the fundamental mechanisms of GAs. A flow chart of canonical GA is shown in Figure 2.1. There in, the GA begins with an initialization step, followed by a repeated sequence of fitness evaluation, selection, crossover and mutation.

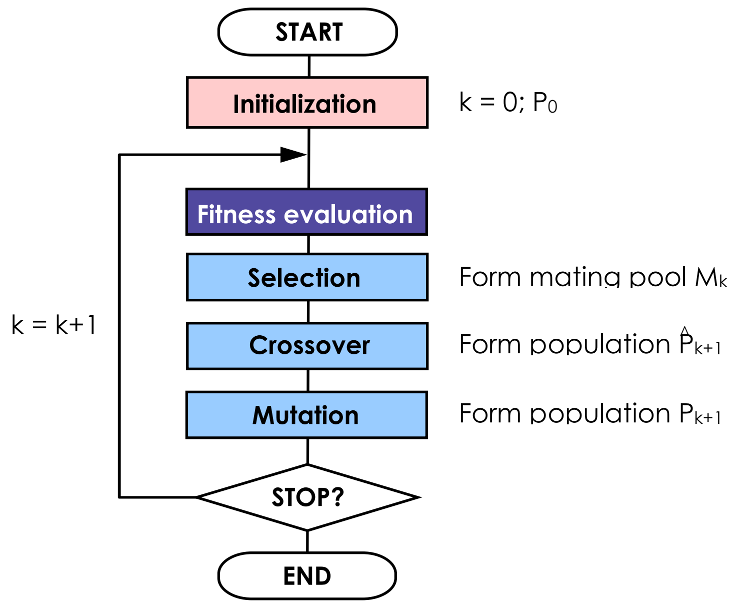


Figure 2.1 Flow chart of a typical GA

Initialization

In the initialization step, initial solutions are randomly generated and encoded into individuals according the predefined representation rule. Binary coding is employed in canonical GAs. The generation number k is set to 0 and the initial population is denoted P_0.

Fitness Evaluation

The fitness value is a figure of merit for an individual. In the fitness evaluation step, each individual is assigned with its fitness value. Generally, higher fitness value corresponds to a more optimal individual.

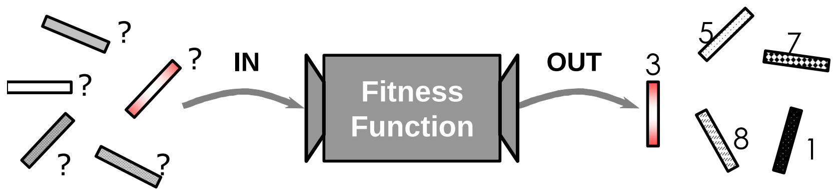


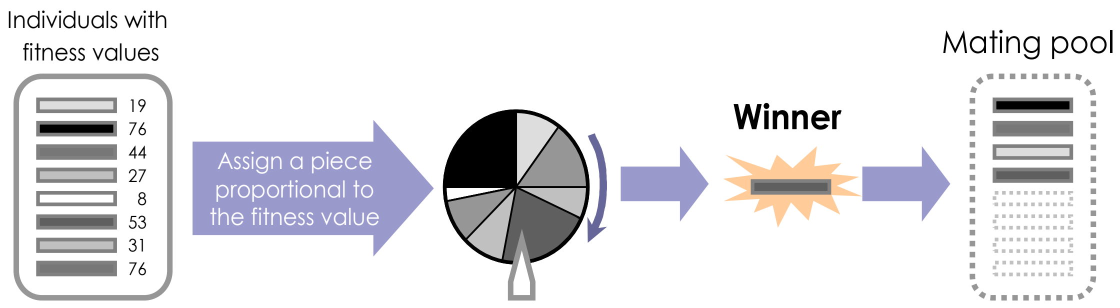
Figure 2.2 Fitness evaluation

Selection

In nature, the individuals that are better suited to the environment are more likely to survive and reproduce. The selection operator emulates this situation by ensuring that individuals with larger fitness values are more likely to survive to reproduce. Among the several different selection methods, the roulette wheel and tournament selection algorithms are commonly used to form a mating pool M_k.

a. Roulette wheel selection

Roulette wheel selection is one of the most popular selection methods. Let's assume that all the individuals are evaluated and assigned with their fitness values. Then one can imagine a roulette wheel with sections whose number is same as the number of individuals and whose areas are proportional to the fitness values of the corresponding individuals. Then the wheel is turned and a chromosome is selected and copied to the mating pool. This process is repeated until the mating pool is full.



Individuals with fitness values
19
76
44
27
8
53
31
76
Assign a piece proportional to the fitness value
Winner
Mating pool

Figure 2.3 Roulette wheel selection

b. Tournament selection

As the name states, two or more individuals are randomly chosen from the population and the one with better fitness value is selected and copied in the mating pool. This method is simpler than the roulette wheel method.

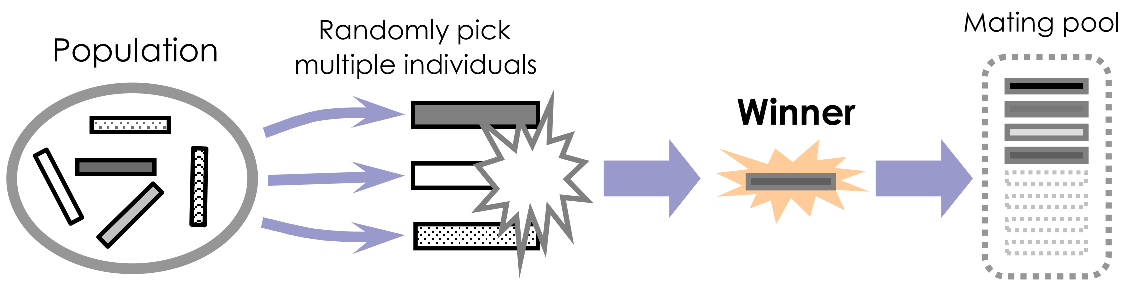


Figure 2.4 Tournament selection

Crossover

Crossover emulates the reproduction of living organs by exchanging gene among the chromosomes. Crossover generates new individuals that share the characteristics of their parents. Crossover is performed on the mating pool M_k to form population $\hat{P}_{k+1}$ as a first step in forming the next generation P_{k+1}. The single-point crossover and the multiple-point crossover operators are list below.

a. Single-point crossover

A crossover point is randomly selected and the genes of the parents are exchanged after the crossover point as depicted in Figure 2.5.

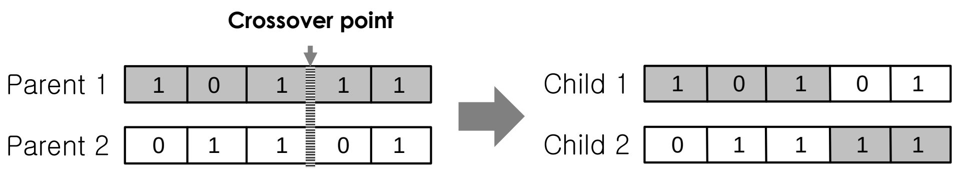


Figure 2.5 Single-point crossover

b. Multiple-point crossover

Several crossover points are randomly chosen and the genes of the parents are exchanged in between the crossover points. Figure 2.6 illustrates two point crossover.

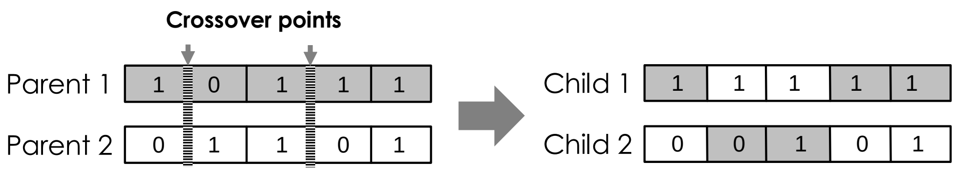


Figure 2.6 Multiple-point crossover

Mutation

In natural evolution, mutation occurs as the result of an error in copying the gene information. As an analogy to this, mutation in GA is a process of changing some genes in chromosomes randomly. The main role of mutation operator is to maintain the diversity of the population.

In the canonical GA using binary representation, mutation operator flips the selected bit value as in Figure 2.7. The mutation operator is applied to $\hat{P}_{k+1}$ which yield P_{k+1}

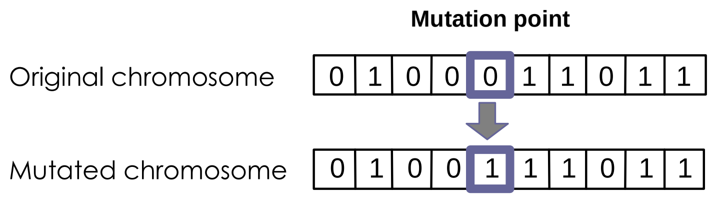


Figure 2.7 Mutation in binary-coded GAs

2.3 Other genetic operators

Selection, crossover and mutation are the primary genetic operators. However, other genetic operators have been developed to improve the performance of GAs. We introduce some of them that are employed in GOSET.

Elitism

Elitism is a mechanism to protect the best individual from being altered and lost by genetic operations. The simplest way to implement elitism is to pass the current best individual to the next population without any genetic operations. By using elitism, it is guaranteed that maximum fitness in the population will never decrease.

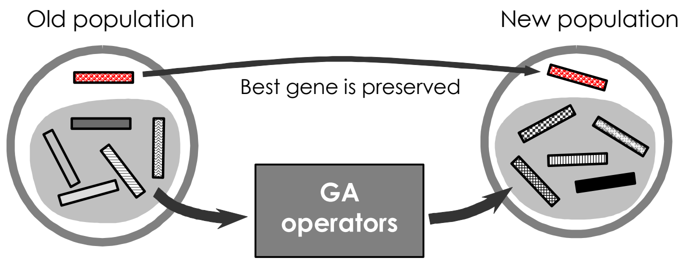


Figure 2.8 Elitism

Migration

This operator works only when multiple-region (or multiple-population) scheme is employed. By setting the number of regions, *n*, greater than one, the population is divided into *n* different populations. Generally, these populations evolve without any interaction. Periodically, some of the individuals are redistributed and move from one region to other region.

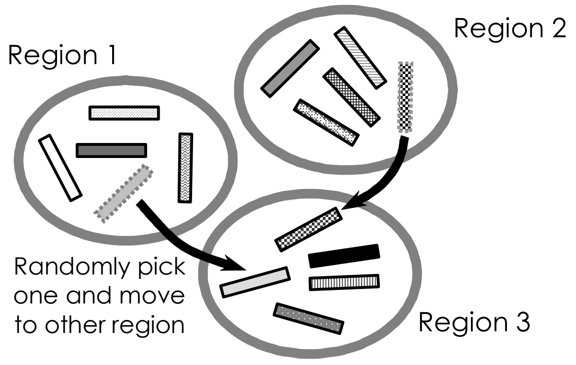


Figure 2.9 Migration operator

Using multiple populations with migration can result in a better chance of finding the global optimum with less computation.

Random search

Random search is a way to extensively explore the neighborhood of the best individual for better solution by random mutation of the best individual. It can help reduce the time for the GA to converge to the optimal solution.

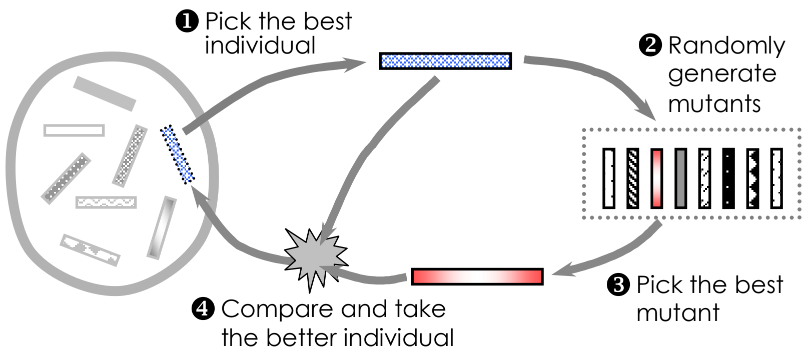
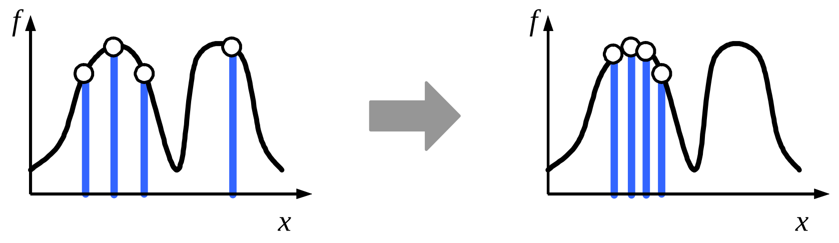

❶ Pick the best individual
❷ Randomly generate mutants
❸ Pick the best mutant
❹ Compare and take the better individual

Figure 2.10 Random search

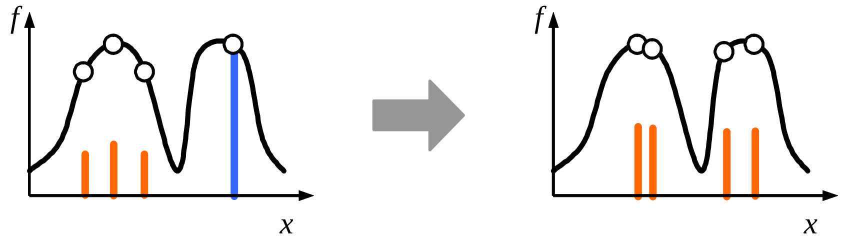
As shown in Figure 2.10, the best individual is randomly perturbed to generate mutants. Then, the fitness values of the generated individuals are evaluated. If the best among the mutants has better fitness value than that of the current best individual, then the current best individual is replaced by the new best individual. Otherwise, the original best individual is placed back to the population.

Diversity Control

For some optimization problems, there are multiple optimal solutions (multi-modal problems). A naive application of GAs can result in convergence of the solutions to one optimal solution. Even in the problem with single optimal solution, it is not desirable for the many solutions exploring the same region in the solution space. Therefore, the diversity control is employed. By using diversity control, the under represented solutions are emphasized and similar solutions are penalized by adjusting their fitness values.



(a) Without the diversity control



(b) With the diversity control

Figure 2.11 Diversity control

Figure 2.11 shows the effect of diversity control. Each circle on the curve represents a solution and its fitness value is shown by the vertical bar below it. Most of the solutions are close to the first optimal solution in (a). With the high probability of selecting a solution near the first optimum, it is likely to end up having all the solutions near the first optimum. However, when the diversity control is used, the fitness function values of the overrepresented solutions in the first optimum are penalized as in (b) and underrepresented solutions in the second optimum are less penalized and have better chance to survive.

Chapter 3

An Overview of Multi-Objective Optimization

GOSET has the capability to perform multi-objective optimizations. A few fundamental notions on multi-objective optimization are introduced in this chapter. A more extensive review, and one which is closely related to GOSET, is set forth in the book *Power Magnetic Devices: A Multi-Objective Design Approach* by S.D. Sudhoff and published by IEEE Press/Wiley.

3.1 Multi-objective optimization problems

Definition

Multi-objective optimization problems involve more than one objective function. Each objective function is to be minimized or maximized. The general form of multi-objective optimization problem can be formally defined as

$$\begin{array}{lll} \text{min/max} & f_m(\boldsymbol{x}), & m = 1, 2, \cdots, M \\ \text{subject to} & g_j(\boldsymbol{x}) \ge 0, & j = 1, 2, \cdots, J \\ & h_k(\boldsymbol{x}) = 0, & k = 1, 2, \cdots, K \\ & \underbrace{x_i^{(L)}}_{\text{lower bound}} \le x_i \le \underbrace{x_i^{(U)}}_{\text{upper bound}}, & i = 1, 2, \cdots, n \end{array}$$

The fundamental difference between single-objective optimization and multi-objective optimization is that in multi-objective optimization problem the desired result is a set of points that describe the best tradeoff between competing objectives rather than a single point representing the extrema of a single objective function.

Pareto optimal solution

In the single-objective optimization problem, the superiority of a solution over other solutions is clearly determined by comparing their objective function values. However, in multiple-objective optimization problem, the goodness of a solution has to be redefined.

For this purpose, the concept of domination is introduced. Suppose there are two solutions x_1 and x_2. The solution x_1 is said to dominate x_2 (or x_2 is dominated by x_1), if the following two conditions are satisfied,

Dominance test conditions

1. The solution x_1 is no worse than x_2 in all objectives.
2. The solution x_1 is strictly better than x_2 in at least one objective.

As an illustration of the concept of domination, let's consider two-objective optimization problem with f_1 and f_2. We want to maximize f_1 and minimize f_2. Assume there are five solutions as in Figure 3.1.

First, compare the solution 1 and the solution 2. The solution 1 is better than the solution 2 for both of the objectives. Hence it is evident that the solution 1 dominates the solution 2.

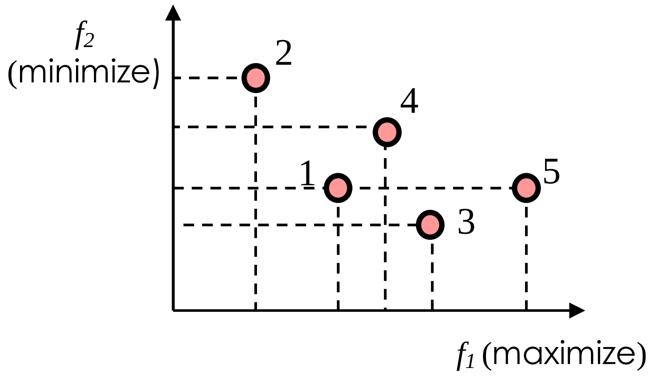


Figure 3.1 Dominance check example

Now look at the solution 1 and solution 5. They have same f_2 values, but solution 5 has bigger f_1 value than solution 1. Thus solution 5 dominates solution 1. As a final example, let's check the dominance between the solution 1 and 4. The solution 4 is better for the first objective function, but the solution 1 is better for the second objective function. As neither solution satisfies the first condition for dominance test, we cannot say that either solution dominates the other.

Given a set of solutions, the **non-dominated solution set** is a set of all the solutions that are not dominated by any members of the solution set.

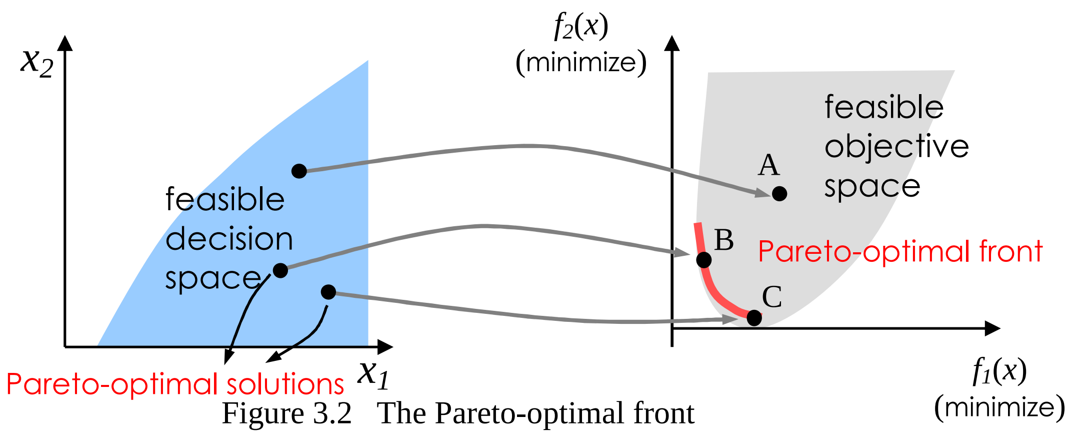


Figure 3.2   The Pareto-optimal front

Each solution in the feasible decision space can be mapped to the feasible objective space. The non-dominated set of the entire feasible search space is called the **Pareto-optimal solution set**. In Figure 3.2, a bold line in the feasible objective space is called the **Pareto-optimal front** that is the set of all the points mapped from the

Pareto optimal solution set. The Pareto-optimal front represents the best possible compromise between conflicting objectives. The Pareto-optimal front is the desired result of the multi-objective optimization.

Diversity control

There are multiple solutions for a given multi-objective optimization problem and any solution in the Pareto optimal solution set can be the best solution. Thus it is required to find not only as many Pareto-optimal solutions as possible, but also as diverse as possible solutions over the Pareto-optimal front.

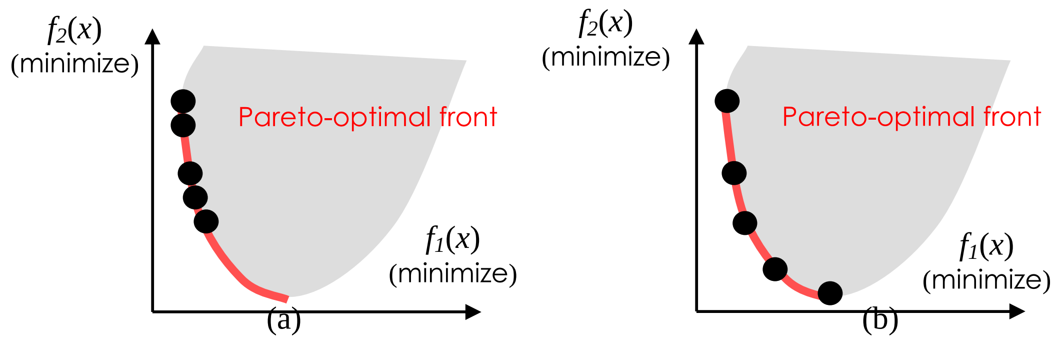


Figure 3.3 Different distributions of solutions

In the Figure 3.3, there are five points in Pareto-optimal front for each case (a) and (b). While the solutions of (a) are concentrated on a specific part of the Pareto-optimal front, those of (b) are evenly distributed over the Pareto-optimal front. There are chances that the most appropriate solution for the given problem exists in the neglected portion of the Pareto-optimal front in case (a). Thus, it is very important to have diverse solutions.

There are several different techniques used to control the diversity of solutions. The interested is referred to [Deb01] or [Car02].

3.2 Genetic algorithms for multi-objective optimization problem

Genetic algorithm utilizes a population of solution candidates. It is possible for the genetic algorithms to find out multiple optimal solutions in one execution. Meanwhile, a series of executions is required to find out multiple solutions in the classical optimization approaches. Therefore, genetic algorithms are highly suitable for solving multi-objective optimization problems.

Schaffer [Sch84] implemented the first multi-objective genetic algorithm in 1984 to find a set of non-dominated solutions. However, it is not until mid 1990's that researchers became actively involved in this area.

Several different multi-objective genetic algorithms have been developed over the years. The followings are some of those.

- Vector Evaluated GA (Schaffer, 1984)
- Non-Dominated Sorting GA (Goldberg, 1989)
- Niched-Pareto GA (Horn et al., 1994)
- Vector-optimized ES ((Frank Kursawe, 1990)
- Multiple objective GA (Fonseca & Fleming, 1993)
- Weighted-Based GA (Hajela and Lin, 1993)
- Random Weighted GA (Murata & Ishibuchi, 1995)
- Distance-based Pareto GA (Osyczka & Kundu., 1995)
- Strength Pareto EA (Zitzler & Thiele., 1998)
- Elitist NSGA (NSGA II) (Deb et al., 2000)
- Pareto-archived ES (Knowles & Corne., 2000)
- Rudolph's elitist MOEA (Rudolph, 2001)

Detailed description of these algorithms can be found in Deb [Deb01].

References

[Car02] Carlos A. Coello Coello, David A. Van Veldhuizen, and Gary B. Lamont, *Evolutionary Algorithms for Solving Multi-objective Problems*, Kluwer Academic Publishers, 2002

[Deb01] K. Deb, *Multi-Objective Optimization using Evolutionary Algorithms,* John Wiley & Sons, Inc., 2001

[Sha84] J. D. Schaffer. *Some Experiments in Machine Learning Using Vector Evaluated Genetic Algorithms.* Ph.D. Thesis, Nashville, TN: Vanderbilt University, 1984

[Sha85] J. D. Schaffer. *Multiple objective optimization with vector evaluated genetic algorithms*. In J. J. Grefenstette, editor, Proc. Int'l Conf. on Genetic Algorithms, pages 93--100, 1985.

Chapter 4

GOSET Data Structures and Algorithm Execution

4.1 Data Structures

A large amount of information is involved in the genetic algorithm execution. To facilitate the information in an organized fashion, GOSET categorizes the information into the following three structures:

MATLAB structure name	Contents
P	Population information
GAP	Genetic Algorithm Parameters
GAS	Genetic Algorithm Statistics

Table 4.1 Data structures

Structure: P

Structure P contains all the information related to the current population. There are 16 fields associated with this structure. Field names and their descriptions are list in the following table.

P.[Field name]	Description
P.fithandle	Handle to the fitness function
P.size	The number of individuals in the population
P.mfit	Unconditioned fitness function values (P.nobj $\times$ P.size)
P.fit	Fitness function values (1 $\times$ P.size)
P.eval	Fitness evaluation flag (1 $\times$ P.size) 0 : fitness is not evaluated 1 : fitness is evaluated
P.age	Age of each individual of the population in generations
P.ngenes	Number of genes in all chromosomes of an individual
P.min	GAP.gd_min
P.max	GAP.gd_max
P.type	GAP.gd_type
P.chrom_id	GAP.gd_cid
P.normgene	Normalized gene values (P.nobj $\times$ P.size)
P.gene	Gene values (P.nobj $\times$ P.size)
P.region	Geographic region (1 $\times$ P.size) of an individual
P.pen	Penalty function (1 $\times$ P.size) which is used for diversity control

Table 4.2 Data structure of the population

Structure: GAP

Structure GAP has all the parameters about genetic operations. There are 67 fields associated with GAP. The description of paremeters and their default values are listed in the Appendix C.

Default values for GAP are defined in gapdefault.m. Thus the user can load the gapdefault and then redefine only the required fields, instead of defining all the fields.

Structure: GAS

The best fitness values, median fitness values, average fitness values, and best chromosomes over the generations are stored in GAS. Current generation number and the number of total objective function evaluations are also stored.

GAS.[Field name]	Description
GAS.cg	Current generation number
GAS.medianfit	The median fitness values of each objective (No. of objectives $\times$ No. of generations)
GAS.meanfit	The average fitness values of each objective (No. of objectives $\times$ No. of generations)
GAS.bestfit	The best fitness values of each objective (No. of objectives $\times$ No. of generations)
GAS.bestgenes	The best gene values for each objective over the generations (No. of genes $\times$ No. of generations $\times$ No. of objectives)
GAS.ne	The number of the total objective function evaluations

Table 4.3 Data structure of GAS

4.2 Algorithm Execution flow

The algorithm execution flow of GOSET is depicted in Figure 4.1. Together with the short description of each step, the related GOSET function names are listed.

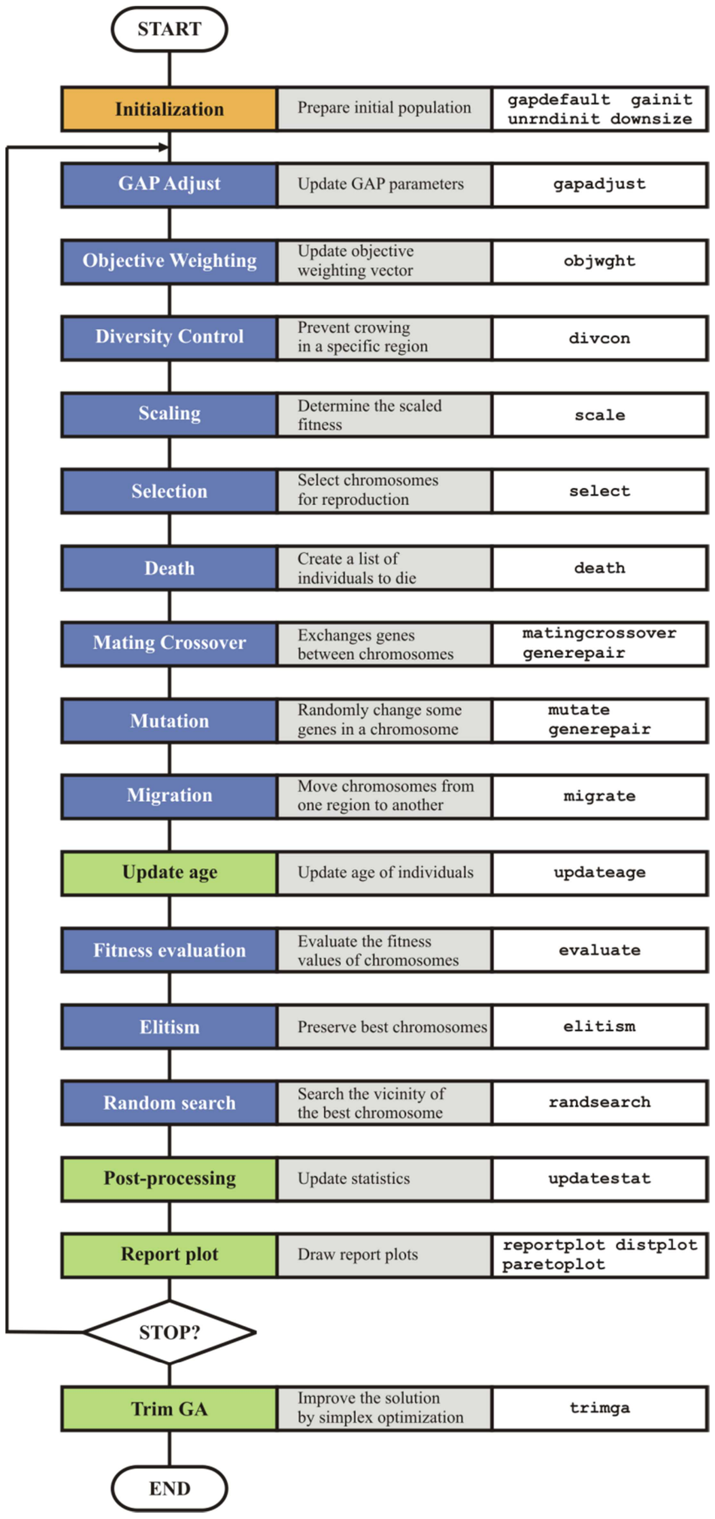

START
Initialization
Prepare initial population
gapdefault gainit unrndinit downsize
GAP Adjust
Update GAP parameters
gapadjust
Objective Weighting
Update objective weighting vector
objwght
Diversity Control
Prevent crowing in a specific region
divcon
Scaling
Determine the scaled fitness
scale
Selection
Select chromosomes for reproduction
select
Death
Create a list of individuals to die
death
Mating Crossover
Exchanges genes between chromosomes
matingcrossover generepair
Mutation
Randomly change some genes in a chromosome
mutate generepair
Migration
Move chromosomes from one region to another
migrate
Update age
Update age of individuals
updateage
Fitness evaluation
Evaluate the fitness values of chromosomes
evaluate
Elitism
Preserve best chromosomes
elitism
Random search
Search the vicinity of the best chromosome
randsearch
Post-processing
Update statistics
updatestat
Report plot
Draw report plots
reportplot distplot paretoplot
STOP?
Trim GA
Improve the solution by simplex optimization
trimga
END

Figure 4.1 Algorithm execution of GOSET

Initialization

In this step, the initial population is randomly generated and data structures P,GAP, and GAS are initialized. When the population does not exist, the initial population of size GAP.ipop is randomly generated. Then the fitness value for each individual is evaluated. The other data structures are also initialized accordingly. If the steady-state population size GAP.npop is smaller than the initial population size GAP.ipop, then the population size is reduced to GAP.npop by discarding inferior chromosomes.

Genetic operators

Various genetic operators act on the current population to generate new population. The detailed descriptions on these operators are in Chapter 5.

Post-processing

Once the new population has been generated, the best fitness value, the average fitness value, and the gene values of the best individual are stored in the data structure GAS.

Report plot

At the completion of the genetic operations, GOSET reports the information on the new population in the gene distribution plot and/or the Pareto plot. In the gene distribution plot, the normalized gene values of the individuals are plotted and also the best fitness value, the average fitness value, the average fitness value, and the worst fitness value over the generations are plotted. In the Pareto plot, the population is plotted in objective function space.

Trim GA

In the single objective optimization problem, this step utilizes the Nelder-Mead simplex algorithm and performs an optimization to improve the best solution at the end of the evolution process.

4.3 Execution of GOSET

When using GOSET from a MATLAB script, GOSET is initiated by gaoptimize.m that has the following syntax format.

```
[fP,GAS]=gaoptimize(objhandle,GAP,D,GAS,iP,GUIhdl)
[fP,GAS]=gaoptimize(objhandle,GAP,D,GAS,iP)
[fP,GAS]=gaoptimize(objhandle,GAP,D)
[fP,GAS]=gaoptimize(objhandle,GAP)

[fP,GAS,bI]=gaoptimize(objhandle,GAP,D,GAS,iP,GUIhdl)
[fP,GAS,bI]=gaoptimize(objhandle,GAP,D,GAS,iP)
[fP,GAS,bI]=gaoptimize(objhandle,GAP,D)
[fP,GAS,bI]=gaoptimize(objhandle,GAP)

[fP,GAS,bI,f] = gaoptimize(@fitfun,GAP,D,GAS,iP,GUIhd1)
[fP,GAS,bI,f] = gaoptimize(@fitnun,GAP,D,GAS,iP)
[fP,GAS,bI,f] = gaoptimize(@fitnun,GAP,D)
[fP,GAS,bI,f] = gaoptimize(@fitnun,GAP)
```

There are 12 arguments for gaoptimze.m which needs to be defined before executing GOSET.

objhandle	objhandle is the handle of the m-file for the fitness function.
GAP	GAP is the structure of genetic algorithm parameters.
D	D is the optional data for the fitness function.
GAS	GAS is the structure of genetic algorithm statistics. If this does not yet exist, pass an empty matrix '[ ]'
iP	iP is the initial population (a structure). If not used, pass an empty matrix '[ ]'

The outputs of gaoptimize.m are fP, GAS, bI and f.

fP	fP is the final population (a structure).
GAS	GAS is the structure of genetic algorithm statistics.
bI	bI is the best individuals or non-dominated solution array.
f	f is the best fitness whose column is a set of fitness values of the individuals corresponding to bI.

It is easy to verify that gaoptimize.m follows the algorithm execution flow given in Figure 4.1. As the gaoptimize.m has a simple modularized structure, it can be modified easily so that the users can experiment with their own routine.

For the detailed description about running GOSET, refer to Chapter 6 that contains step-by-step illustrations of using GOSET for several different optimization problems.

Chapter 5

GOSET genetic operators

In this section, the genetic operators used in GOSET are explained in detail.

5.1 GAP adjust

The GAP adjust routine is responsible for changing the mutation and elitism parameters as a function of generation. Each mutation parameter (such as standard deviation) and some elitism parameters have an initial and final value, and the GAP adjust algorithm sets this as a linear interpolation between the initial and final values based on generation number.

5.2 Objective weighting

In the multi-objective optimization problem, there are more than one fitness values for each individual. `objwght.m` randomly generates a normalized weighting vector to be used for scalarization of the objective function values.

In the multi-objective optimization problem (`P.nobj` > 1), it is possible to use only one objective function value for fitness evaluation. If the objective function number to be used is specified in `GAP.fp_obj`, then the output weight vector owv has all zero values except for the element corresponding to the objective function specified by `GAP.fp_obj`.

5.3 Diversity control

Four different diversity control algorithms are available to maintain the diversity of the solutions in GOSET. These routines generate a fitness weight value for each individual. These fitness weight values constitute the fitness penalty vector (`P.pen`) that is used for determining an aggregated fitness `P.fit` in the scaling process. Individuals with many other individuals close to them are assigned a small fitness weight value (thereby reducing the effective fitness) and those with small number of neighboring individuals are assigned with fitness weight value near unity (thus, penalizing the fitness less).

The diversity control can be applied to either the parameter (or decision) space or the fitness function (or objective) space. For the diversity control in the parameter space, use `GAP.dc_spc = 1` and `GAP.dc_spc = 2` is for the diversity control in the fitness function space.

a. Diversity control algorithm 1 (`GAP.dc_alg = 1`)

In this approach, the number of neighboring individuals of each individual is counted. The neighboring individuals are those within the threshold distance which is determined as

$$\text{Threshold distance} = (\text{mean distance between points}) \times \alpha ,$$

where α = (`GAP.dc_mnt`+rand$\times$(`GAP.dc_mxt`-`GAP.dc_mnt`)).

Then the fitness weight of an individual is the reciprocal of the counted number of individuals. Figure 5.1 illustrates this method with three examples.

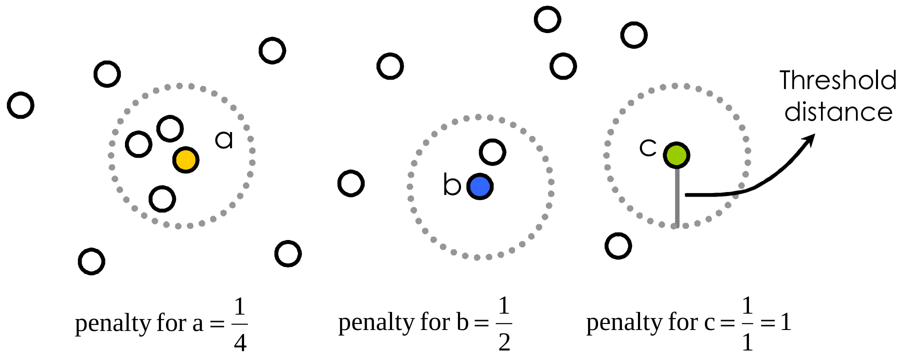

a
b
c
Threshold distance
penalty for a = $\frac{1}{4}$
penalty for b = $\frac{1}{2}$
penalty for c = $\frac{1}{1}$ = 1

Figure 5.1 Illustration of diversity control method 1 in 2D space.

While very systematic, one major drawback of this approach is the necessity of evaluating the distance between all the individuals which requires a computation time proportional to the square of the population.

b. Diversity control algorithm 2 (`GAP.dc_alg = 2`)

This approach is based on the idea that individuals with similar gene values have similar weighted sum of their gene values for any weight vector. In this method, first an integer weight vector is generated at random, where the element value come from the integer set {1, 2, … `P.ngenes`}. Then the weighted sum of the normalized genes for each individual is evaluated. Individuals with similar weighted sum values are grouped and put into bins based on the weighted sum. The total number of bins are randomly chosen from {(`GAP.dc_mnb` × `GAP.fp_npop`), … (`GAP.dc_mxb` × `GAP.fp_npop`)}. For the individuals in a specific bin, their fitness penalty weights become the reciprocal of the number of individuals in that bin.

Since it is possible that two individuals with drastically different gene values have similar weighted sum for some weight vector, this procedure is repeated `GAP.dc_ntr` times and the largest fitness penalty weight is chosen as the final fitness penalty weight for each individual. This reduces the chance of assigning an individual a smaller fitness penalty weight than appropriate.

Figure 5.2 shows an illustration of the diversity control algorithm mentioned above. For example, if we look at the bin No. 1, there are two individuals. So the penalty value for the individuals in bin No. 1 is ½, and likewise, the penalty value of the individual in bin No. 5 is ¼.

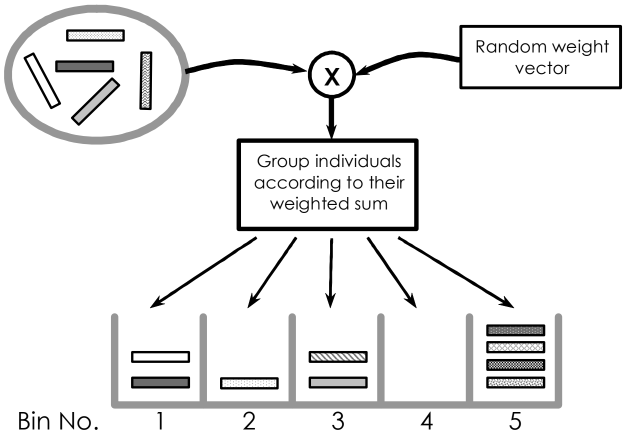


Figure 5.2 Illustration of diversity control method 2.

Although this approach is not as systematic as the first approach, the computation time is proportional to the population size, not the square of the size.

c. Diversity control algorithm 3 (`GAP.dc_alg = 3`)

The idea of this diversity control algorithm is similar to the diversity algorithm 1. Instead of using the count of solutions in a neighborhood, the sum of infinity norm between a solution and all the other solutions is used to determine the penalty value. The fitness penalty weight for *k*'th individual is express as

$$P_{pen}^{k} = \frac{1}{\sum_{i=1} \exp\left(-\frac{d_{i,k}}{d_c}\right)},$$

where $d_{i,k}$ is the infinity norm between *k*'th and *i*'th individual, that is, the maximum absolute gene difference between *k*'th and *i*'th individual and d_c is the distance constant (`GAP.dc_dc`) which controls the size of the neighborhood. If a small d_c is used, then the effective size of the neighborhood is also small. Thus only the solutions with many neighboring solutions that are very close to them are penalized severely and most of other solutions are not penalized. As the d_c increases, the effective size of the neighborhood increases and the penalty level also increases.

The fitness penalty weight in the algorithm 3 can take continuous value rather than discrete value as in the algorithm 1. As with the algorithm 1, the distance evaluation between all the individuals requires a computation time proportional to the square of the population.

d. Diversity control algorithm 4 (`GAP.dc_alg = 4`)

This approach is identical to the diversity control algorithm 3. However, only GAP.dc_nt individuals among the population are randomly selected for the distance evaluation. The random selection of the individuals is performed for each different individual. This reduces the computational load at the cost of some inaccuracy in the distance measurement. The fitness penalty weight for *k*'th individual is express as

$$P_{pen}^{k} = \frac{1}{1 + \dfrac{\text{Number of population in the region}}{\texttt{GAP.dc\_nt}} \displaystyle\sum_{i=1}^{\texttt{GAP.dc\_nt}} \exp\left(-\frac{d_{i,k}}{d_c}\right)},$$

where $d_{i,k}$ is the infinity norm between *k*'th and *i*'th individual, d_c is the distance constant (`GAP.dc_dc`)

5.4 Scaling

In the early stage of the evolution, if there are few individuals with very large fitness values, then these strong individuals will dominate the entire population very quickly which can lead to convergence to some local optimum without thorough exploration of the search space. This is called as premature convergence. Towards the end of the evolution, when the population is almost converged with most of the individuals sharing similar fitness values, then the competition among individuals is weak and the evolution process slows down. As a remedy to both these problems, scaling can be employed to maintain the appropriate evolution pressure throughout the evolution process. Scaling is also useful in the multi-objective optimization problems that have different scales in the objective functions.

As the first step in the scaling operation, the fitness values are scaled using one of the six scaling methods. After scaling, all negative fitness values are clipped to zero, and then the objective function weight vector (`GAP.owv`) is applied to scalarize the fitness values (`P.mfit`) in the multi-objective optimization. Finally, the penalty vector (`P.pen`) is applied and the scalarized the fitness values are penalized to yield the aggregated fitness values (`P.fit`) that are used in the selection operation.

Several different scaling methods are available in GOSET. Options include no-scaling, offset scaling, standard linear scaling, modified linear scaling, mapped linear scaling, sigma truncation, and quadratic scaling. These methods are described below.

a. No scaling (`GAP.sc_alg = 0`)

Scaling is not applied and the actual fitness value is used as shown in Figure 5.3. This option is primarily intended for fitness functions that have been carefully constructed so that no scaling is necessary.

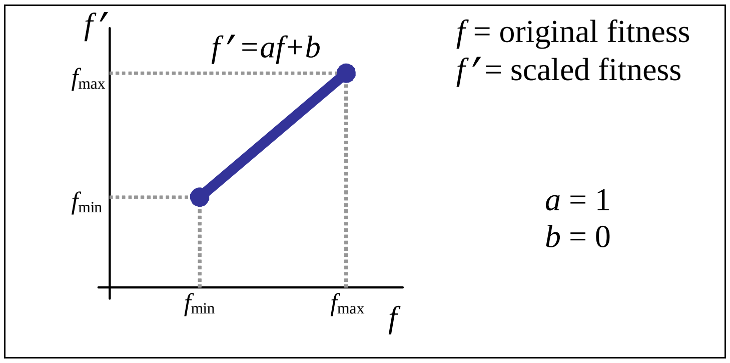


Figure 5.3 No scaling

b. Offset scaling (`GAP.sc_alg = 1`)

In this method, fitness values are mapped linearly such that the minimum fitness value is mapped to 0 and the maximum value is mapped to $f_{max} ! f_{min}$.

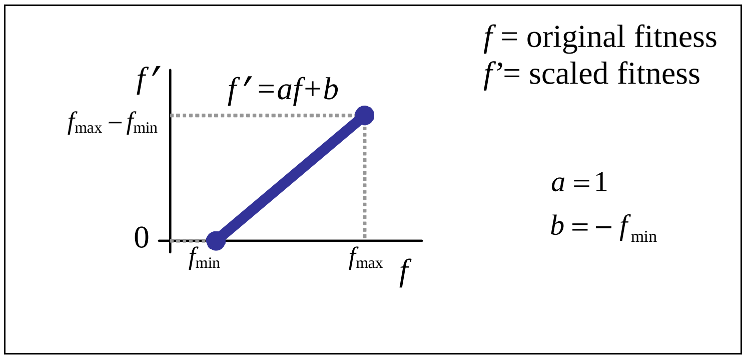
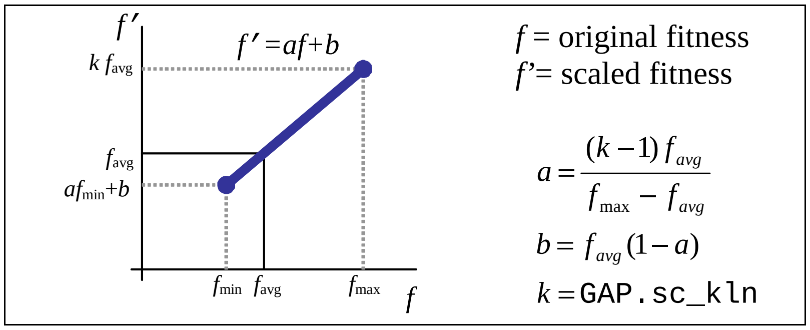


Figure 5.4 Offset scaling

c. Standard linear scaling (`GAP.sc_alg = 2`)

In this method, a linear scaling is used in such a way that the average fitness is not modified and the maximum fitness is `GAP.sc_kln` times the average fitness value.

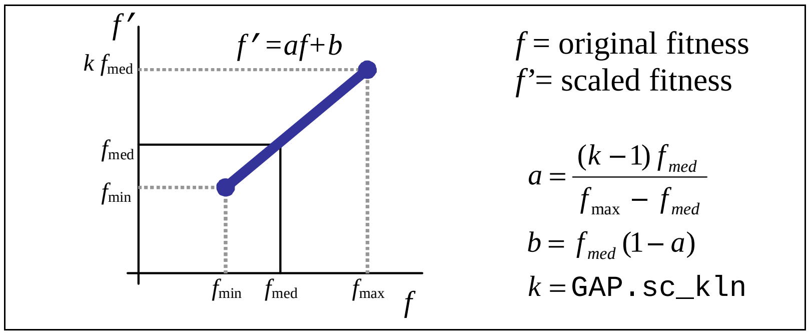


$$a = \frac{(k-1) f_{avg}}{f_{\max} - f_{avg}}$$
$$b = f_{avg}(1-a)$$
$$k = \texttt{GAP.sc\_kln}$$

Figure 5.5 Standard linear scaling

d. Modified linear scaling (`GAP.sc_alg` = 3)

In this method, a linear scaling is applied in such a way that the median fitness is not modified and the maximum fitness is GAP.sc_kln times the median fitness value.

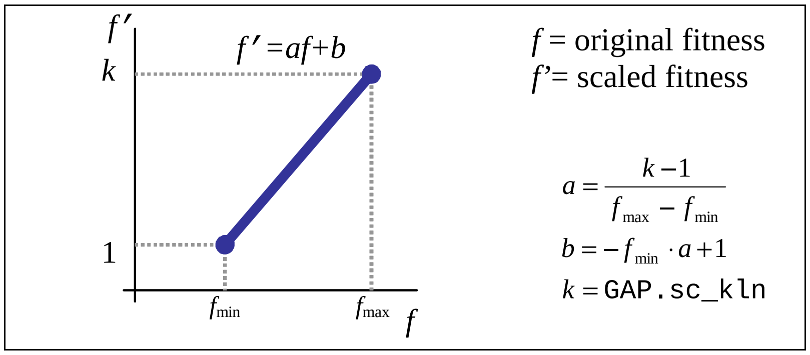


$$a = \frac{(k-1) f_{med}}{f_{\max} - f_{med}}$$
$$b = f_{med}(1-a)$$
$$k = \texttt{GAP.sc\_kln}$$

Figure 5.6 Modified scaling

e. Mapped linear scaling (`GAP.sc_alg` = 4)

This method is another linear scaling that maps the maximum fitness to GAP.sc_kln and the minimum fitness to 1.



$$a = \frac{k-1}{f_{\max} - f_{\min}}$$
$$b = -f_{\min} \cdot a + 1$$
$$k = \texttt{GAP.sc\_kln}$$

Figure 5.7 Mapped linear scaling

f. Sigma truncation (`GAP.sc_alg` = 5)

In the sigma truncation method, all the fitness values smaller than (f_{avg} - `GAP.sc_cst` $\times f_{std}$), where f_{avg} is the average fitness value and the f_{std} is the standard deviation of the fitness values, are mapped to negative values and therefore disregarded later by clipping to zeros. It is useful when there are a few individuals with very small fitness value and most individuals have large fitness values.

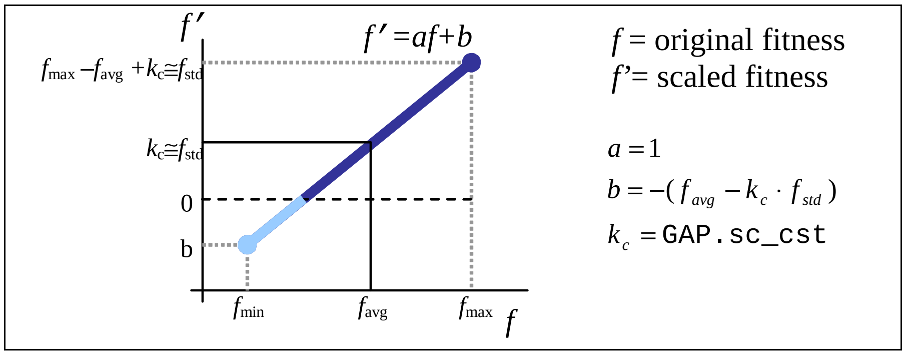
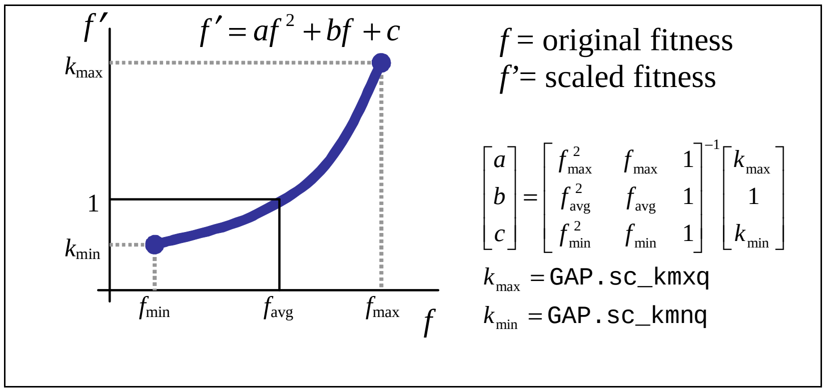


Figure 5.8 Sigma truncation scaling

g. Quadratic scaling (`GAP.sc_alg` = 6)

This algorithm emphasizes the large fitness value and deemphasizes the small fitness value. The maximum fitness value is mapped to `GAP.sc_kmxq`, the average fitness value to 1, and the minimum fitness value to `GAP.sc_kmnq`. The quadratic scaling is the only nonlinear scaling method in GOSET.



f'
$f' = af^2 + bf + c$
k_{max}
1
k_{min}
f_{min}
f_{avg}
f_{max}
f
f = original fitness
f'= scaled fitness
$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} f_{max}^2 & f_{max} & 1 \\ f_{avg}^2 & f_{avg} & 1 \\ f_{min}^2 & f_{min} & 1 \end{bmatrix}^{-1} \begin{bmatrix} k_{max} \\ 1 \\ k_{min} \end{bmatrix}$
k_{max} = GAP.sc_kmxq
k_{min} = GAP.sc_kmnq

Figure 5.9 Quadratic scaling

5.5 Selection

The selection operators choose individuals from the population to constitute a mating pool for reproduction. When the multiple regions are used, selection operations are confined to each region. For each region, the selection operator picks same number of chromosomes as those in the current region and moves them to the mating pool for that region.

There are two pre-defined selection operators that the user can choose from. They are roulette wheel selection and tournament selection.

a. Roulette wheel selection

Each individual is assigned with the selection probability that is proportional to the aggregate fitness value (`P.fit`). Then individuals are chosen based on the selection probability. It is more likely that the better individual is chosen and copied to the mating pool which mimics principle of the survival of the fittest.

b. Tournament selection

`GAP.sl_nts` number of individuals are randomly chosen from the population and their aggregate fitness values (`P.fit`) are compared. Then the individual with the best fitness value is selected to be in the mating pool.

Illustrations of these selection operators are in the section 2.2.

Custom algorithm

In the case that the user wants to use his/her own selection algorithm, the custom algorithm handle `GAP.sl_cah` can be defined as long as the algorithm follows certain format. The details regarding the custom algorithm can be found in the Appendix B.

5.6 Death

Death operator determines which individual is to die and replaced by the children. The followings are possible options for the death operators.

a. Replacing parents

Parents are replaced by their own children.

b. Random selection

The parents to be replaced are randomly chosen.

c. Tournament on fitness

The parent to be replaces is determined via the tournament based on the aggregate fitness value. `GAP.dt_nts` number of parents are randomly chosen for a tournament and the one with worst aggregate fitness value is marked for death.

d. Tournament on age

The parent to be replaces is determined via the tournament based on the age. Among the randomly chosen `GAP.dt_nts` number of parents, the oldest one is marked for death.

e. Custom algorithm

User defined custom death algorithm is used. The custom function handle is assigned to `GAP.dt_cah`. Refer to the Appendix B for the details on the format of the custom algorithm.

f. Random algorithm

If this option is selected, the death algorithm is randomly chosen among the first four death algorithms at each generation.

5.7 Mating and crossover

There are three different crossover operators in GOSET. All the crossover operation is performed on the normalized gene values and the actual gene values are updated based on the crossovered normalized gene values.

These crossover operations are followed by gene repair process for illegal genes. That is, if a gene value lies outside of the allowed range $[0,1]$ after the crossover operation, that gene value is automatically fixed using the generepair routine.

a. Single point crossover

This crossover operator is similar to the crossover operator in binary-coded GAs. In multiple chromosome setting, single point crossover occurs in each chromosome. The following example shows a single point crossover operation on individuals with two chromosomes.

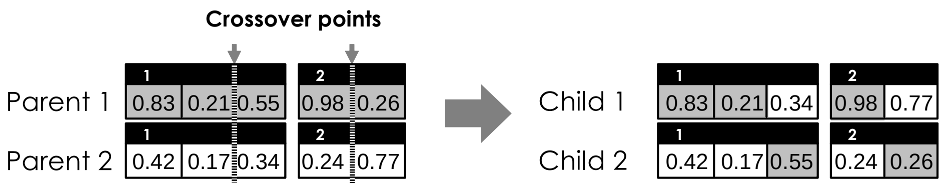


Figure 5.10 Single point crossover with two chromosomes

b. Simple blend crossover

In the simple blend crossover, the children are generated from the weighted sum of their parents. It is implemented so that the gene values of the two children have same distance from the average value of the gene values of the parents. Figure 5.11 illustrates how simple blend crossover works. Filled circles represent the gene values of parents positioned at *p* and *q* respectively, and white circles represent those of children. The gene values of children can take any values between (3*p*-*q*)/2 and (3*q*-*p*)/2 and they are equally distanced from the center of *p* and *q*.

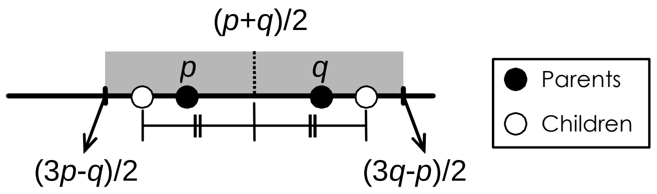


Figure 5.11

Depending on whether the each gene value in a chromosome is blended using same ratio or each gene value is blended independently, there are scalar simple blend crossover and vector simple blend crossover.

Scalar simple blend crossover

In the scalar simple blend crossover operation, each gene position has different ratio of blending. For example, two parents

Parent 1 = [0   0.8   0.3] and Parent 2 = [1 0.2 0.5]

can generate

Child 1 = [0.25   0.95   0.4] and Child 2 =   [0.75 0.05   0.4]

via scalar simple blend crossover. The first gene values moved 25% of their distance towards the average value of them. The second gene, -25%. And the third gene, 50%.

Vector simple blend crossover

In the vector simple blend crossover operation, all the genes are blended using same ratio. For example, two parents

Parent 1 = [0   0.8   0.3] and Parent 2 = [1 0.2 0.5]

can generate

Child 1 = [0.25   0.65   0.35] and Child 2 =   [0.75 0.35   0.45].

For all three genes, the parent gene values are blended in such a way that they moved 25% of the distance between them towards their average values.

c. Simulated binary crossover

As the name suggests, the simulated binary crossover operator mimics the behavior of the single-point crossover operator in binary-coded genetic algorithm. Detailed description of the simulated binary crossover operator is beyond the scope of this manual and only the basic concepts are introduced here. Interested readers are referred to [p109, Deb01]

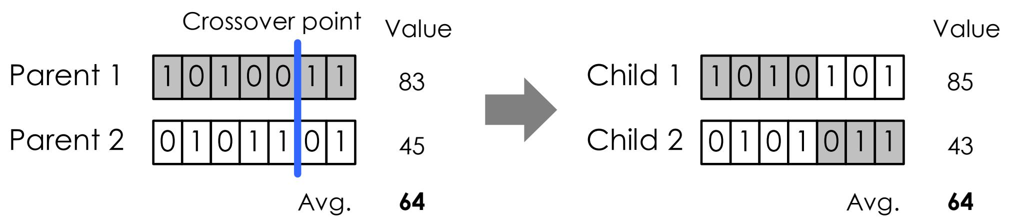


Figure 5.12 Single point crossover example

Figure 5.12 illustrates an example of the single point crossover operation on the binary chromosomes. Note that the average values are same before and after the crossover operation. Hence, the amount of increase in one chromosome is same as the decrease in another chromosome and the children are equally distanced from the center point of the parents.

Each point of the chromosome has the same probability to be selected as a crossover point. And the crossover in the lower bit results in children closer to the parents point. Thus the values of children are more like to be near the values of parents. With these investigations, the single point crossover can be simulated in real-coded genetic algorithms by using the probability density for the children as in Figure 5.13.

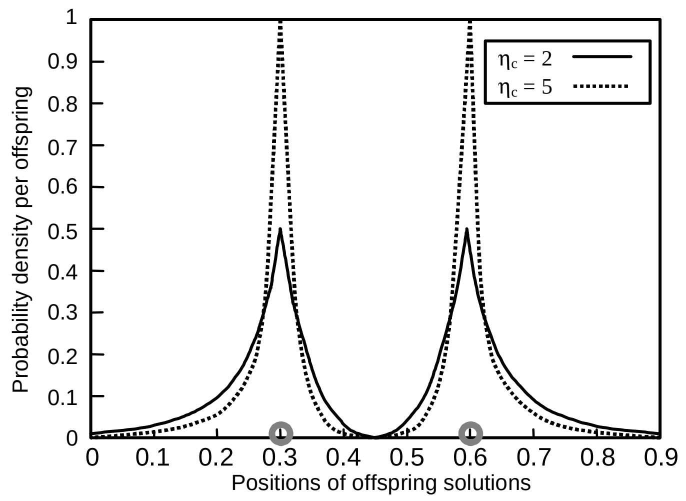


Figure 5.13 Simulated binary crossover

In Figure 5.13, it is assumed that the parents are positioned at 0.3 and 0.6. The distribution index η_c is a non-negative real number that controls the spread of the children. If the distribution index η_c is large, the probability of generating children that are closer to the parents are higher. As the distribution index η_c becomes smaller, it is allowed to create solutions that are far from the parents.

As in the simple blend crossover, there are *scalar simulated binary crossover* and *vector simulated binary crossover* depending on whether the each gene value is crossovered using same ratio or each gene value is crossovered independently.

d. Random crossover

When `GAP.mc_alg` is set to 6, GOSET chooses a mating crossover algorithm randomly among the five methods described above. They are

- Single point crossover
- Scalar simple blend crossover
- Vector simple blend crossover
- Scalar simulated binary crossover
- Vector simulated binary crossover.

For every `GAP.mc_gac` generation, the crossover algorithm changes randomly.

5.8 Mutation

The mutation operators of GOSET can be categorized into three types as described in this section. All the mutation operations are applied to the normalized gene value and the actual gene values are updated based on the mutated normalized gene values.

a. Total mutation

With the probability of `GAP.mt_ptgm`, each gene value can be replaced by a new randomly generated gene value within the prescribed range that is defined by `P.max` and `P.min`. For the integer type gene, the gene value takes any integer value within the allowed range.

In the following figure, let us assume that the real-typed third gene is mutated. As the mutation is applied to the normalized gene values, each gene has a value between 0 and 1.

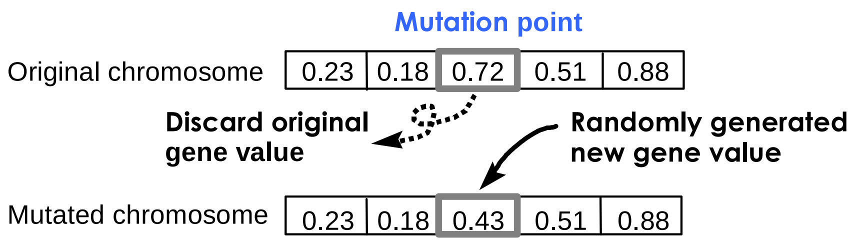


Figure 5.14 Total mutation

b. Partial mutation

Two types of partial mutations are employed. They are the relative partial mutation and the absolute partial mutation. Integer genes are not involved in the partial mutations.

These mutation operations are followed by gene repair process (generepair) for fixing illegal genes.

Relative partial mutation

With the probability of `GAP.mt_prgm`, each gene value is perturbed by certain fraction of the current gene value. The amount of perturbation is obtained using a Gaussian random variable with standard deviation of σ_{rgm}(`GAP.mt_srgm`).

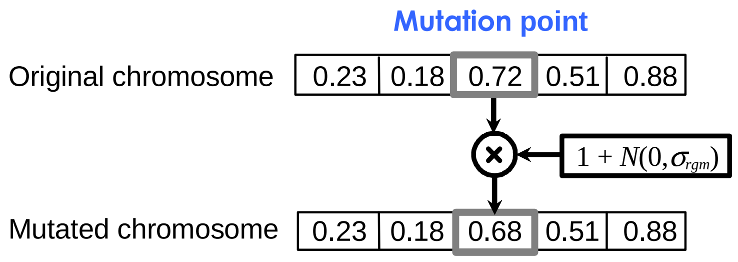


Figure 5.15 Relative partial mutation

The figure 5.15 illustrates a relative partial mutation on the third gene, when the Gaussian random variable $N(0,\sigma_{rgm})$ has a value -0.055.

Absolute partial mutation

With the probability of `GAP.mt_pagm`, each gene value is added with a Gaussian random variable with zero mean and standard deviation of σ_{agm}(`GAP.mt_sagm`).

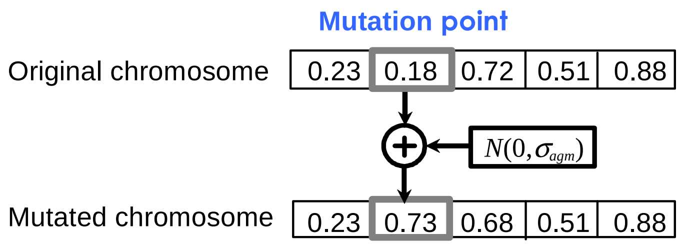


Figure 5.16 Absolute partial mutation

The figure 5.16 illustrates a relative partial mutation on the third gene, when the Gaussian random variable $N(0,\sigma_{agm})$ has a value 0.55.

c. Vector mutation

Vector mutation is very similar to the partial mutation. However, the vector mutation changes the each and every gene of the individual undergoing mutation. Integer genes do not participate in the vector mutations.

These mutation operations are also followed by gene repair process for illegal genes as in the partial mutation.

Relative vector mutation

Each individual undergoes relative vector mutation with the probability of `GAP.mt_prvm`. Every gene value of the individual is perturbed by certain fraction of the current gene value. The amount of perturbation is obtained using

$$\text{Random vector} = v_{dir} \cdot N(0,\sigma_{rvm})$$

where v_{dir} is a normalized random vector $(P_{ngenes} \times 1)$ specifying the direction of perturbation and $N(0,\sigma_{rvm})$ is a Gaussian random variable with mean 0 and standard deviation `GAP.mt_srvm`.

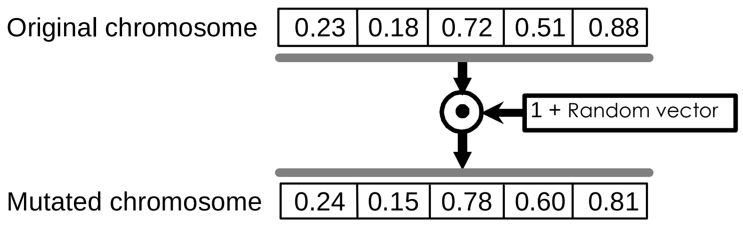


Figure 5.17 Relative vector mutation

The figure 5.17 illustrates a relative vector mutation when the random vector is given as

[0.03 –0.15 –0.08 0.18 –0.08].

Absolute vector mutation

Each individual undergoes absolute vector mutation with the probability of `GAP.mt_pavm`. If an individual mutates, each and every gene value of the individual is added with a random vector

$$\text{Random vector} = v_{dir} \cdot N(0,\texttt{GAP.mt\_savm}),$$

where v_{dir} is a normalized random vector $(P_{ngenes} \times 1)$ specifying the direction of perturbation and $N(0,\sigma_{avm})$ is a Gaussian random variable with mean 0 and standard deviation `GAP.mt_savm`.

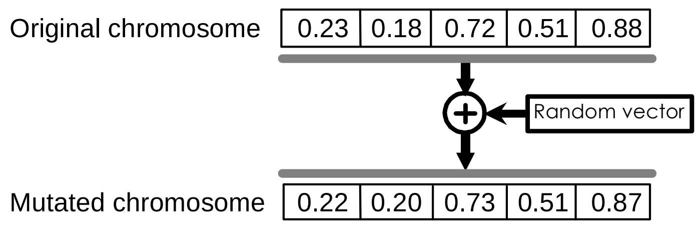


Figure 5.18 Absolute vector mutation

The figure 5.18 illustrates an absolute vector mutation with the random vector

$$[\,-0.01 \quad 0.02 \quad 0.01 \quad 0.0 \quad -0.01\,].$$

d. Integer mutation

Integer mutation is applied only to integer genes. Each gene is mutated to a randomly generated integer within the allowed range with the probability of `GAP.mt_igm`.

Each mutation related parameters is dynamically updated within the initial value and final value defined by the user. For example, the total mutation probability `GAP.mt_ptgm` at each generation is calculated as

$$\texttt{GAP.mt\_ptgm} = \texttt{GAPic.mt\_ptgm0} \times (1-\beta) + \texttt{GAPic.mt\_ptgmf} \times \beta$$

where $\beta = \frac{\text{current generation number}}{\text{total number of generation}}$. Thus the total mutatio probability starts from the initial value `GAP.mt_ptgm0` and varies gradually to the final value `GAP.mt_ptgmf`.

5.9 Gene repair

Sometimes the gene values generated by the matingcrossover operator or the mutation operator are infeasible and they fall outside of the specified range. In that case, they need to be repaired to have valid gene value. Two different gene repair algorithms are available to maintain the feasibility of the solutions in GOSET. This routine is called within the matingcrossover and mutation operators.

a. Hard limiting algorithm 1 (`GAP.gr_alg` = 1)

When the processed gene value lies outside of the allowed range, i.e. [0, 1], the hard limiting method maps a gene value to the nearest boundary value. For example, if a resultant gene value is 1.2, it is adjusted to 1, and if it is -0.4, it is adjusted to 0.

b. Ring mapping algorithm 2 (`GAP.gr_alg` = 2)

When the processed gene value lies outside of the allowed range, i.e. [0, 1], the ring-mapping maps a gene value to the modulus after division by 1. For example, if a resultant gene value is 1.2, it is adjusted to 0.2, and if it is -0.1, it is adjusted to 0.9.

5.10 Migration

Migration is meaningful only when there are multiple regions defined, that is, `GAP.mg_nreg` > 1. This operator selects some individuals in the population and moves them to different regions. Each individual is migrated with the probability of `GAP.mg_pmig`. The parameter `GAP.mg_tmig` determines the frequency of applying the migration operator and the migration interval is randomly chosen between 0.5×`GAP.mg_tmig` and 1.5×`GAP.mg_tmig`. For example, if `GAP.mg_tmig` is set to 4, then the possible migration intervals are 2, 3, 4, 5, and 6 generations.

5.11 Fitness evaluation

In this step, the fitness values of all individuals are evaluated. `GAP.ev_bev` determines whether to evaluate the fitness of an individual at a time (`GAP.ev_bev` = 0) or to evaluate the fitness of the entire population at once (`GAP.ev_bev` = 1).

With GOSET 2.6, a new feature is the ability to utilize parallel processing, so that multiple cores are used to calculate the fitness of the population. Note that this requires the use of MATLAB's Parallel Processing Toolbox. Setting `GAP.ev_pp` = true invokes this option. The number of evaluation groups is set to `GAP.ev_npg` which has a default value of 12, and should be set to the number of cores being used.

When an individual is moved from the previous generation without any change, the fitness value of the individual does not change and there is no need to evaluate the fitness again. In this case, GOSET can evaluate only the unevaluated individuals by setting `GAP.ev_are` = 0. Setting `GAP.ev_are` = 1 will force GOSET evaluate all individuals.

Evaluation of the fitness usually requires only the gene values of the individual (`P.gene`). If the optional data D is specified for the `gaoptimze` function call, then the optional data is also passed to the fitness function. On top of this, other information like age(`P.age`), previous fitness function values(`P.mfit`), and region(`P.region`) can be send to the fitness function by setting `GAP.ev_ssd` = 1.

There are many different ways to define a valid fitness function for a given optimization problem. One fundamental rule is that the better gene should have more positive fitness function value than those of inferior genes.

As an example, let's look at the minimization problem of Powell function. The Powell function [CHO96] is described as

$$f(x_1, x_2, x_3, x_4) = (x_1 + 10x_2)^2 + 5(x_3 - x_4)^2 + (x_2 - 2x_3)^4 + 10(x_1 - x_4)^4 .$$

It is the minimization problem and the smaller function value is better. Hence, we can simply take the negative of *f*(***x***) as the fitness function. The following is the fitness evaluation routine in m-file for the problem of minimizing Powell function with -*f*(***x***) as the fitness function.

```
function fv = powell(x)

x1 = x(1);
x2 = x(2);
x3 = x(3);
x4 = x(4);

f = (x1 + 10*x2)^2 + 5*(x3 - x4)^2 + (x2 - 2*x3)^4 + 10*(x1-x4)^4;

fv = -f;
```

The Powell function always takes the nonnegative value, and thus the inverse of *f*(***x***) can be another valid fitness function. A small positive value is added to the denominator to avoid the possible singularity at the optimum point. In this case, the last line of above m-file is replaced by the following line.

```
fv = 1/(0.001+f);
```

5.12 Elitism

Elitism is a device to insure that the fittest individual in a population is preserved unless a better fit individual is found.

Single-objective optimization case

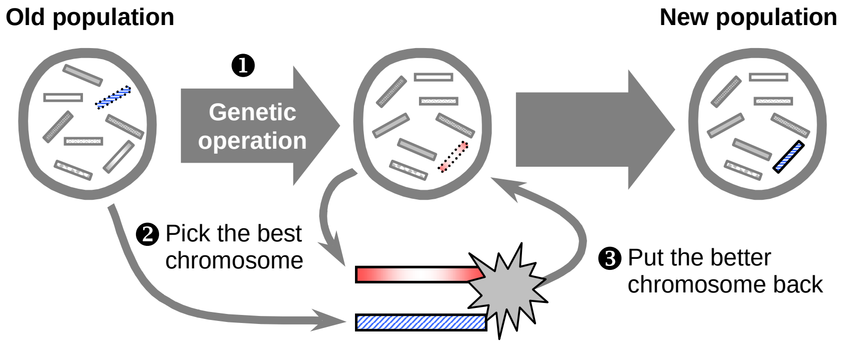


Figure 5.19 Elitism for the single-objective optimization

Elitism in the single objective optimization is straightforward. The best individual of the population P_k and the best individual of the population after the genetic operations performed are compared. The better of the two becomes a member of the next population. As this operation is confined within a region, the best one of each region is preserved for the multi-region case.

Even in the multi-objective optimization case, there are cases that it is desirable to use only one specific fitness function value for elitism. This can be done by setting `GAP.fp_obj` to the number indicating a specific objective function. The elitism, then, performs in the same way as with the single-objective optimization using only one objective function value.

Multi-objective optimization case

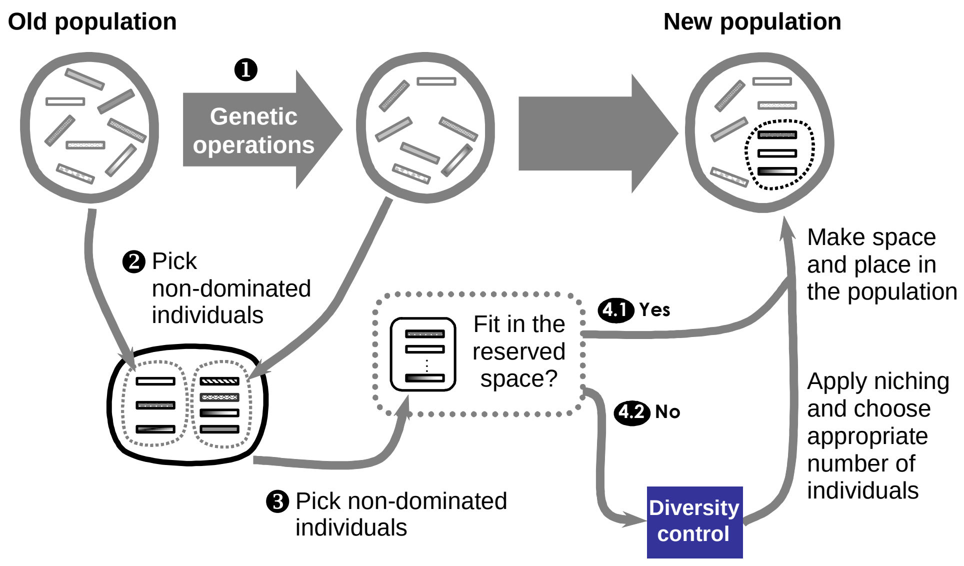

Old population
New population
❶
Genetic operations
❷ Pick non-dominated individuals
Fit in the reserved space?
4.1 Yes
Make space and place in the population
4.2 No
Apply niching and choose appropriate number of individuals
❸ Pick non-dominated individuals
Diversity control

Figure 5.20 Elitism for the multi-objective optimization

In the multi-objective optimization case, the objective of elitism is to preserve non-dominated solutions. Thus, it is necessary to retain multiple individuals.

First, the old population and the modified population in the same region are combined, and the non-dominating solutions are found.

As the number of non-dominated solution can increase, the number of non-dominated solutions is limited to

$$\text{Maximum No. of non-dominated solutions} = \texttt{GAP.el\_fpe} \times (\text{No. of individuals in the region}).$$

If the reserved space for the non-dominating solutions is enough for the non-dominating solutions just found, then some individuals corresponding to dominated solutions are randomly removed from the population and replaced by non-dominated solutions. If the reserved space cannot accommodate all the non-dominated solutions, then the appropriate number of non-dominated solutions are chosen using a diversity control algorithm and placed in the population for the next generation.

As in the mutation operator, the elite fraction is dynamically adjusted within the initial value and final value defined by the user. That is, the fraction of elite GAP.el_fpe at each generation is calculated as

$$\texttt{GAP.el\_fpe} = \texttt{GAP.el\_fpe0} \times (1-\beta) + \texttt{GAP.el\_fpef} \times \beta$$

where $\beta = \frac{\text{current generation number}}{\text{total number of generation}}$. Thus the elite ratio in the population starts from the initial value GAP.el_fpe0 and varies gradually to the final value GAP.el_fpef.

5.13 Random search

Random search operator is specialized for a local search. It can reduce the convergence time significantly near the optimum point. In the initial stage of the evolution when the active exploration is desirable, it is unnecessary to apply random search. The parameter `GAP.rs_fgs` specifies the point of starting the random search. If `GAP.rs_srp` = 0.2, then the random search is inactive for the first 20 percent of entire generation. At each generation with the active random search, the random search occurs with the probability given by `GAP.rs_fea`.

Given the best solution, random search operator generates mutants of the best chromosome. Mutants are generated in the same way as the relative vector mutation based on `GAP.rs_srp` and the absolute vector mutation with `GAP.rs_sap`. The relative vector mutation is chosen with the probability of `GAP.rs_frp` and the absolute vector mutation is chosen with the probability of (1–`GAP.rs_srp`). The number of the generated mutants is determined by the parameter `GAP.rs_fps` that specifies the fraction of the total population size, that is

The number of mutants = `GAP.rs_fps` $\times$ Size of the population.

Then the best solution among the mutants is found and this solution replaces the existing best only if this solution is better than the existing best solution.

5.14 Trim GA

The trimga operator uses the Nelder-Mead simplex algorithm to perform a deterministic optimization using the best individual from a GA as a starting point. The goal is to find a better solution in the vicinity of the obtained GA solution. The trimga only works with single-objective optimization problems. Gene range constraints are enforced by subtracting infinitity from the fitness function when the gene range goes outside of the prescribed limits. This is a stand alone routine and is not the part of the evolution process.

A sample call is

```
[x,f] = trimga(GAP,P,D)
```

or

```
[x,f] = trimga(GAP,P)
```

where the inputs are the genetic algorithm parameter structure GAP the population structure P and optional data structure D, and where the outputs are x the revised solution, and f the revised fitness function value.

References

[CHO96] E. K. P. Chong and S. H. Żak, *An Introduction to optimization*, Wiley-Interscience, 1996

[Deb01] K. Deb, *Multi-Objective Optimization using Evolutionary Algorithms*, John Wiley & Sons, Inc., 2001

Chapter 6

Tutorial Lessons

In this section, some optimization problems are considered with step-by-step guidance to familiarize the users with GOSET. Each problem is solved using both the command line approach and the GUI approach.

6.1 Rosenbrock’s banana function

Rosenbrock’s function [p.55, CHO96] is a real-valued function given in the following:

$$\boxed{f(x_1,x_2)=100(x_2-x_1^2)^2+(1-x_1)^2}$$

We want to find the minimizer of the function $f(x_1, x_2)$. Rosenbrock’s function and its level sets are depicted in Fig. 6.1 and Fig. 6.2. Due to the shape of level sets that resemble bananas, it is also referred to as the banana function. The global optimizer of Rosenbrock’s function is at (1, 1) where the function has its value 0.

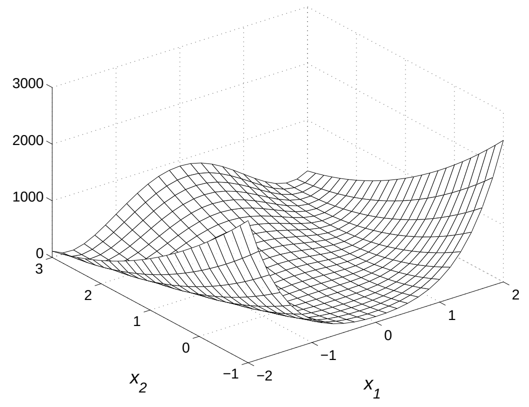


Figure 6.1 Rosenbrock’s function

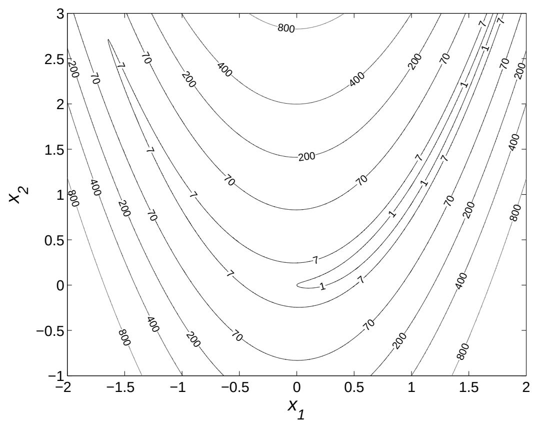


Figure 6.2 Level sets

Before using GOSET, the fitness function for the given problem needs to be defined as an mfile. There are many different ways to define a valid fitness function. As it is a minimization problem and the Rosenbrock’s function value is non-negative, one simple way is to take the inverse as the fitness function. A small positive value is added to the denominator to prevent the singularity of the fitness function value at the minimizer. The following mfile `bananafit.m` defines a fitness function for Rosenbrock’s function.

```
function f = bananafit(x)
% Rosenbrock's banana function

% Banana function
f1 = 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;

% Fitness function
f = 1/(0.001 + f1);
```

The above mfile is located in the folder ‘`Rosenbrock`’ under ‘`GOSET Examples`’ folder.

With the fitness function defined, we are now ready to use GOSET to find the minimizer of the Rosenbrock's function.

We will use a script `banana.m` to use GOSET to solve Rosenbrock's problem. This script is listed below

```
% Optimization of Rosenbrock's Problem

% Initialize the parameters
GAP = gapdefault;                               % Line 1

% Define gene parameters
%                  x1     x2
% gene              1      2
GAP.gd_min  = [    -2     -1    ];              % Line 2
GAP.gd_max  = [     2      3    ];              % Line 3
GAP.gd_type = [     2      2    ];              % Line 4
GAP.gd_cid  = [     1      1    ];              % Line 5

% Execute GOSET
[P,GAS,best] = gaoptimize(@bananafit,GAP);   % Line 6
```

First of all, GAP and other parameters related to the population need to be determined. This is done using the call to `gapdefault` in Line 1 (as indicated on the right side). For detailed information regarding the default setting of GAP, refer to `gapdefault.m`.

The values of x_1 and x_2 become the gene values, and their minimum, maximum values are defined in Lines 2 and 3, respectively. The types of the genes are assigned in Line 4. As can be seen, both gene types are 2 indicating linear. Finally the chromosome numbers of each genes are assigned in Line 5. As can be seen, both genes are on chromosome 1.

The optimization engine is invoked in Line 6. Typing 'banana' at the MATLAB prompt in the directory rims the script.

As the default value for the report level is set to `GAP.rp_lvl = 1`, a plot will appear to show the normalized objective function values and the fitness function values as the GOSET evolves over the generations. Figure 6.3 is the report plot after 100 generations.

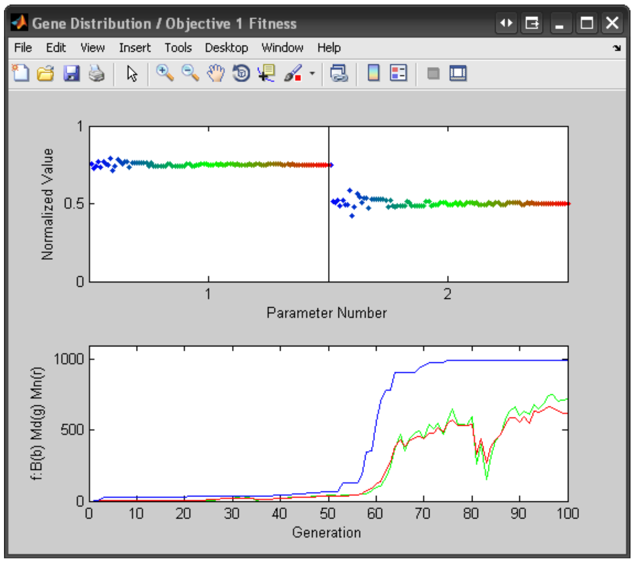


Figure 6.3 Report plot for Rosenbrock's function

There is also the text report displayed in the MATLAB main window. For this example the text report is:

```
Statistics for generation 1
Best fitness = 16.3028
Mean fitness = 0.19982
Median fitness = 0.0074076
Number of evaluations = 100
.
.
.

Statistics for generation 100
Best fitness = 999.947
Mean fitness = 433.3743
Median fitness = 377.9158
Number of evaluations = 6361
```

The generation number and the best, mean, and median fitness values together with the number of evaluations are reported.

The best gene values can be found by checking content of `best`.

```
>> best

ans =

    1.0002
    1.0004
```

The resultant minimizer found by GOSET turned out to be very close to the actual minimizer (1, 1).

6.2 Tanaka Problem

Problem description

One of the most important features of GOSET is the capability to solve multi-objective optimization problems. As a multi-objective optimization problem, Tanaka problem [TAN95] is considered in this section. The Tanaka problem is a constrained optimization problem with two objectives to be minimized:

$$\begin{aligned}
&\min\, f_1(x_1,x_2) = x_1 \\
&\min\, f_2(x_1,x_2) = x_2 \\
&\text{subject to } C_1(x_1,x_2) = x_1^2 + x_2^2 - 1 - 0.1\cos\left(16\arctan\tfrac{x_1}{x_2}\right) \geq 0, \\
&\qquad C_2(x_1,x_2) = (x_1-0.5)^2 + (x_2-0.5)^2 \leq 0.5, \\
&\qquad 0 \leq x_1 \leq \pi, \\
&\qquad 0 \leq x_1 \leq \pi.
\end{aligned}$$

In this problem, the variable space is also the objective space. The feasible objective space and the Pareto-optimal front are shown in Figure 6.4.

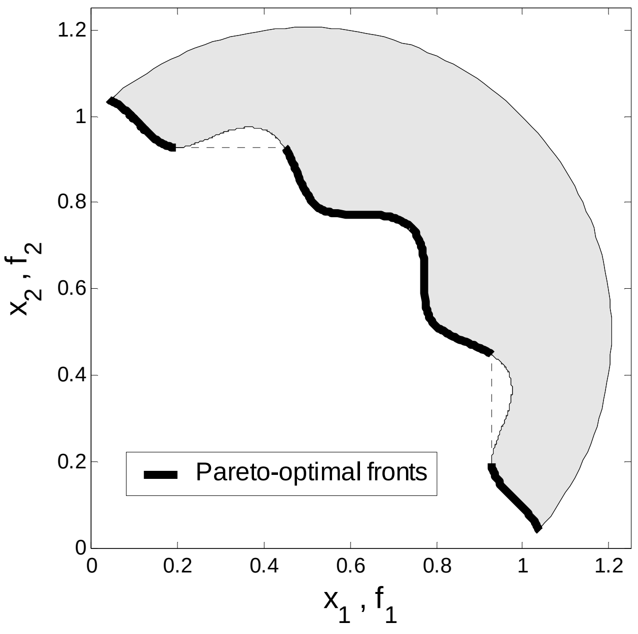


Figure 6.4 Feasible objective space and Pareto-optimal front of Tanaka problem.

In the first step, the fitness function for the given problem needs to be defined in an m-file. There are two objectives to be minimized and they all have positive values. And the fitness function values are defined to be the negative of the objective function values. Infeasible solutions are assigned with the value -10 to reduce the chance of surviving in the population. The following mfile, `tanaka_fit.m`, defines a fitness function for Tanaka problem.

```
function [f] = tanaka_fit(x)

C1 = x(1)^2+x(2)^2-1-0.1*cos(16*atan(x(1)/x(2))) >= 0;
C2 = (x(1)-0.5)^2+(x(2)-0.5)^2 <= 0.5;

if C1 && C2
    f(1,1) = -x(1);
    f(2,1) = -x(2);
else
    f(1,1) = -10;
    f(2,1) = -10;
end
```

The mfile script to perform the optimization to titled `tanaka.m` and is listed on the next page. Therein, the first step is to initialize the genetic algorithm parameters which is accomplished with the call to `gapdefault(2,0,200,200)` which is used to define default GAP for the problem with two objectives, multi-objective optimization, the population size of 200 and generations number of 20. Next, some of the plotting parameters are adjusted in order to provide real-time plotting of the Pareto-optimal front. Next, the gene ranges, types, and chromosome numbers are established. Then `gaoptimize` is called to perform optimization. The remaining code plots the solution, as shown in Fig. 6.5.

```
% Initialize the parameters
GAP = gapdefault(2,0,200,200);

% Plotting parameters
GAP.op_list = [];               % objectives list for objective plots
GAP.pp_list = [ 1, 2];          % gene list for Pareto plot
GAP.pp_sign = [-1,-1];          % sign of fitness for each objective
                                % (1=positive,-1=negative)
GAP.pp_axis = [0 1.25 0 1.25];  % axis limits for Pareto plot

% Gene setup
%                      x1          x2
% gene                  1           2
GAP.gd_min  = [    0.0001      0.0001     ];
GAP.gd_max  = [        pi          pi     ];
GAP.gd_type = [         2           2     ];
GAP.gd_cid  = [         1           1     ];

% Perform optimization
[P,GAS,best]= gaoptimize(@tanaka_fit,GAP);

% Plot final solutions
figure(2)
plot(best(1,:),best(2,:),'x');
axis([0 1.25 0 1.25]);
xlabel('f_1');
ylabel('f_2');
axis square;
title('Pareto Front')
```

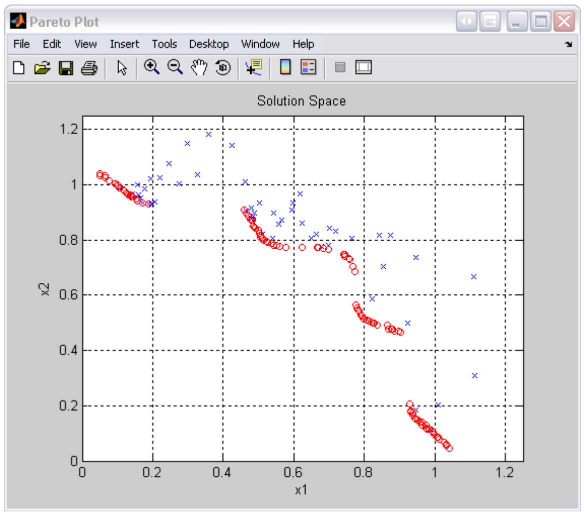


Figure 6.5 Pareto plot for Tanaka problem

The distribution plot is turned off by using null matrix ‘`[]`’ for `GAP.op_list`, and only the Pareto plot is displayed as in Figure 6.5. The final population after 200 generation is plotted on the feasible objective space with the non-dominated solutions in black circles in Figure 6.6. Comparison of this figure with Figure 6.4 demonstrates the performance of GOSET with respect to multi-objective optimization problems.

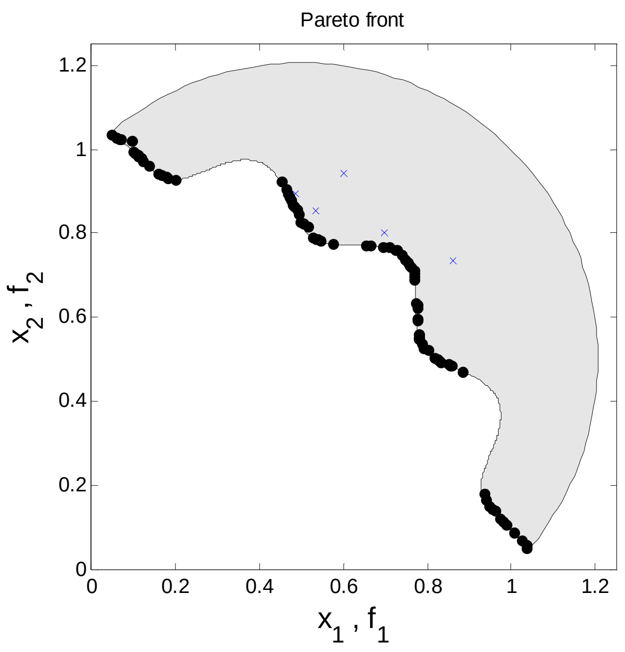


Figure 6.6 Final population plot with the non-dominated solutions in black circles

6.3 Power diode curve fitting

Problem description

In this section, GOSET is applied to power diode curve fitting. The diode of interest is a part of Fuji Electric 6MBI 30L-060 which is commonly used for an inverter for motor drivers and AC-DC servo drive amplifiers. The configuration of Fuji 6MBI 30L-060 is shown in Figure 6.7 and its characteristics are listed in Table 6.1.

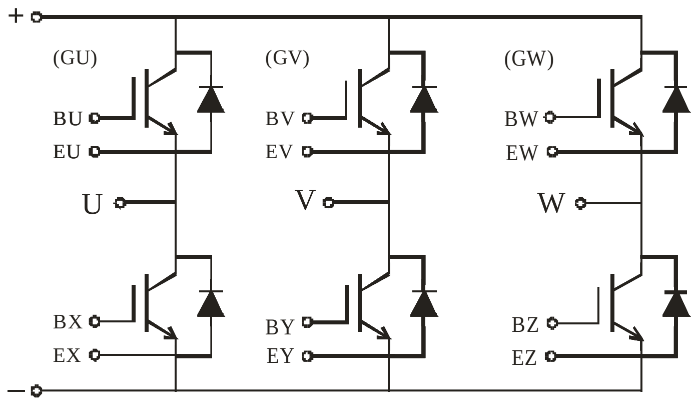
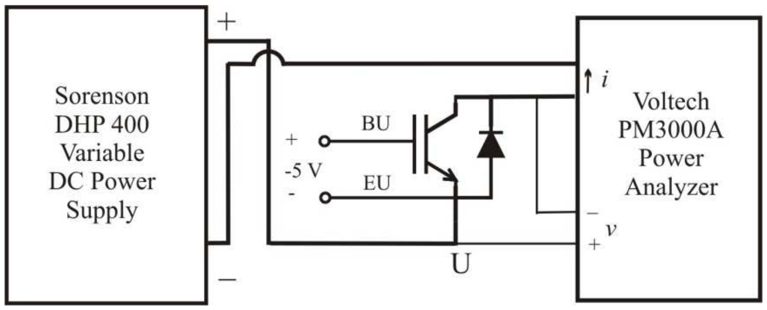


Figure 6.7 Circuit schematic of the IGBT module

Fuji 6MBI 30L-060 Device Characteristics	
Description	**Rating**
Collector-Emitter Voltage	600V
Gate-Emitter Voltage	±20V
Collector Current – Continuous	30A
Collector Current – 1ms Pulse	60A
Maximum Power Dissipation	120W
Operating Junction Temperature	150°C
Thermal Resistance – IGBT Junction to Case	1.04°C/W (Max)
Thermal Resistance – Diode Junction to Case	2.01°C/W (Max)

Table 6.1 IGBT module device characteristics

The voltage versus current (V-I) curve of the power diode is measured using the hardware configuration shown in Figure 6.8


Sorenson DHP 400 Variable DC Power Supply
+
−
BU
EU
+
-5 V
-
U
i
v
−
+
Voltech PM3000A Power Analyzer
Figure 6.8 Hardware test configuration for diode *v*-i characteristic

The *v*-*i* data set measured at 36 points is listed in Table 6.2 and is depicted in Figure 6.9. Using the measured *v*-*i* data set (v_k, i_k), $k = 1, . . . ,n$, a model of the power diode is going to be developed. It is assumed that the voltage is expressed as the function of the current as

$$v = ai + (bi)^c$$

where the parameter *a*, *b*, and *c* are to be identified using GA.

Voltage (V)	Current (A)	Voltage (V)	Current (A)	Voltage (V)	Current (A)
0.3130	0	1.2178	6.0040	1.5160	18.0660
0.4145	0	1.2548	7.0870	1.5358	19.0300
0.5154	0	1.2844	8.0360	1.5527	19.9480
0.6140	0.0400	1.3122	9.0000	1.5716	21.0100
0.7120	0.1495	1.3412	10.0520	1.5885	22.0200
0.8056	0.3915	1.3654	11.0290	1.6065	23.0800
0.8942	0.8345	1.3888	11.9900	1.6229	24.0400
0.9649	1.4103	1.4134	13.0640	1.6389	25.0000
1.0405	2.3180	1.4345	14.0280	1.6565	26.0500
1.1092	3.4680	1.4570	15.0860	1.6720	27.0000
1.1722	4.8340	1.4768	16.0510	1.6880	27.9700

Table 6.2 Measured voltage and current of the power diode

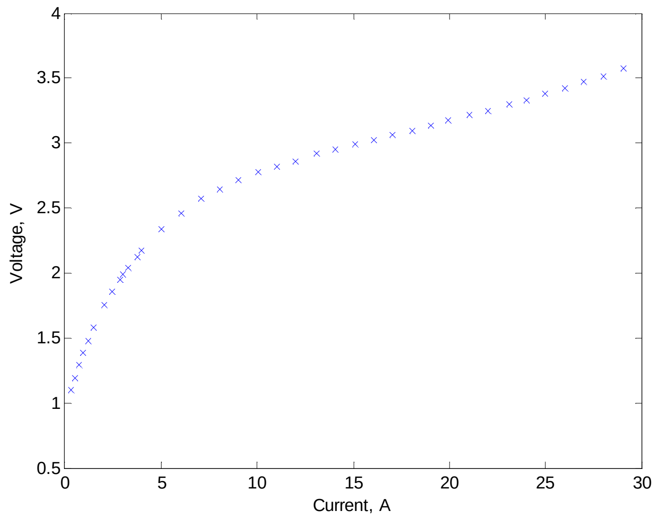


Figure 6.9 Voltage versus current for power diode

A fitness function candidate is

$$f(a,b,c) = \frac{1}{\sum_{k=1}^{n} \left| 1 - \frac{a i_k + (b i_k)^c}{v_k} \right| + 10^{-3}}$$

where v_k and i_k are measured voltage and current in k^{th} point. This fitness function is coded to mfile `IGBT_fit.m` as listed on the next page. In this case, the script to evaluate the fitness serves two roles. First, if called with two arguments (the gene values in the parameter vector and the data in the data structure), it returns the fitness. If called with an additional argument (a figure number), the fitness function also provides post-processing support in the sense that it provides some graphical output.

```
function fitness = igbt(parameters,data,fignum)

% assign genes to parameters
a = parameters(1);
b = parameters(2);
c = parameters(3);

vpred = a*data.i+(b*data.i).^c;
error = abs(1-vpred./data.v);
fitness = 1.0/(1.0e-6+mean(error));

if nargin>2

   figure(fignum);
   Npoints=200;
   ip=linspace(0,max(data.i),Npoints);
   vp=a*ip+(b*ip).^c;
   plot(data.i,data.v,'bx',ip,vp,'r')
   title('Voltage Versus Currrent');
   xlabel('Current, A');
   ylabel('Voltage, V');
   legend({'Measured','Fit'});

   figure(fignum+1);
   Npoints=200;
   plot(data.i,data.v.*data.i,'bx',ip,vp.*ip,'r')
   title('Power Versus Currrent');
   xlabel('Current, A');
   ylabel('Power, V');
   legend({'Measured','Fit'});

end
```

A script to perform the optimization titled `igbt.m` is listed on the next page.

```
% Power diode V-I curve fitting

% Voltage-current measuerment data for power diode in a
Fuji 6MBI30L-060
load igbt_data.mat

% plot the raw data
figure(1);
plot(igbt_data.i,igbt_data.v,'x');
xlabel('Current, A');
ylabel('Voltage, V');
title('Static IGBT Characteristics at 25 C');

% Initialize the parameters
GAP = gapdefault(1,1,500,100);
GAP.mg_nreg = 4;
GAP.mg_tmig = 20;
GAP.mg_pmig = 0.05;

% Define gene parameters
%          min  max  chrm chrm     par
%          val  val  type  id       #   description
GAP.gd = [ 1e-8 1e+0   3  1; ...  % 1  a
           1e-6 1e+3   3  1; ...  % 2  b
           1e-3 1e+0   3  1];     % 3  c

% Execute GOSET
[P,GAS,bestx] = gaoptimize(@igbt_fit,GAP,igbt_data);

% Plot the measured V-I data and the estimated V-I curve
igbt_fit(bestx,igbt_data,4);
```

As can be seen, the first two actions in the script are to load and plot the i-v data, which is stored in the structure `igbt_data`. Next, the genetic algorithm parameters are set with a call to gapdefault. Single-objective optimization with a population size of 500, and 100 generations is specified. The migration parameters are then adjusted (mostly to illustrate how GAP parameters can be manipulated).

The next step in the script is setting forth the gene parameters. This script illustrates a second syntax for accomplishing this. This form is more commonly used that in the first two examples, particularly if there are a large number of parameters.

The optimization engine `gaoptimize` is then called. Finally, the fitness function is called as post-processing so that the results obtained are graphically illustrated.

Executing the script `igbt`.m yields

```
Performing single-objective optimization of objective 1 using
a population size of 500 over 100 generations
Statistics for generation 1
Best fitness = 5.6548
Mean fitness = 1.5385
Median fitness = 1.6202
Number of evaluations = 500

.
.

Statistics for generation 100
Best fitness = 8.8502
Mean fitness = 8.7337
Median fitness = 8.8465
Number of evaluations = 32814

This single-objective optimization of objective 1 using a
population size of 500 over 100 generations

Trimming best answer ...
TRIMGA increased fitness of objective 1 from 8.8502 to 8.8502
TRIMGA violates 0 gene range limits
```

The elements of *bestx* yield

$$a = 4.89 \cdot 10^{-7}$$
$$b = 4.64$$
$$c = 0.257$$

The plots of measured data and the estimated curve are shown in Figure 6.10. The estimated curve fits very closely to the measured data.

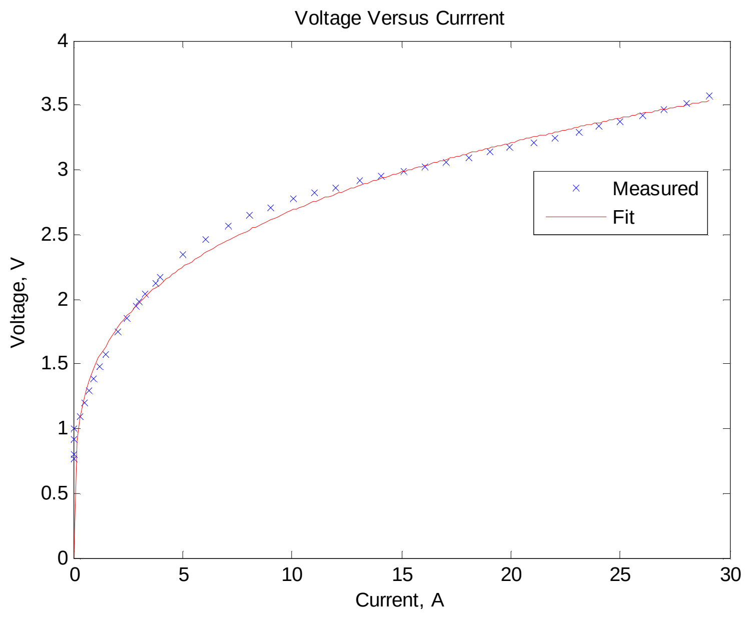


Figure 6.6 Plot of measured data and the estimated curve using GOSET

6.4 Transfer function fitting

Problem description

In this section, GOSET is employed to estimate the transfer function given the transfer function values. The transfer function values are admittances looking into the d-axis of brushless DC motor.

The admittances measured at 60 different frequencies are listed in Table 6.3 and plotted in Figure 6.11.

Freq.(Hz) f_k	Admittance Y_k	Freq.(Hz) f_k	Admittance Y_k	Freq.(Hz) f_k	Admittance Y_k
20	0.2754 + 0.2059i	224	0.0126 + 0.0533i	2516	0.0013 + 0.0062i
23	0.2470 + 0.2106i	253	0.0108 + 0.0478i	2839	0.0012 + 0.0056i
25	0.2299 + 0.2116i	286	0.0094 + 0.0428i	3203	0.0011 + 0.0050i
29	0.1992 + 0.2100i	322	0.0082 + 0.0385i	3615	0.0010 + 0.0045i
32	0.1794 + 0.2066i	364	0.0072 + 0.0345i	4079	0.0009 + 0.0040i
37	0.1517 + 0.1987i	410	0.0063 + 0.0310i	4603	0.0008 + 0.0036i
41	0.1336 + 0.1914i	463	0.0056 + 0.0278i	5195	0.0008 + 0.0033i
47	0.1117 + 0.1798i	523	0.0049 + 0.0249i	5862	0.0007 + 0.0029i
53	0.0948 + 0.1685i	590	0.0044 + 0.0223i	6615	0.0006 + 0.0027i
59	0.0815 + 0.1579i	666	0.0040 + 0.0201i	7465	0.0006 + 0.0024i
67	0.0679 + 0.1452i	751	0.0035 + 0.0180i	8424	0.0005 + 0.0021i
76	0.0565 + 0.1326i	848	0.0032 + 0.0162i	9506	0.0005 + 0.0019i
85	0.0480 + 0.1219i	957	0.0029 + 0.0145i	10728	0.0004 + 0.0017i
96	0.0401 + 0.1108i	1079	0.0026 + 0.0130i	12106	0.0004 + 0.0016i
109	0.0333 + 0.0999i	1218	0.0023 + 0.0117i	13661	0.0004 + 0.0014i
123	0.0280 + 0.0903i	1375	0.0021 + 0.0105i	15416	0.0003 + 0.0012i
138	0.0238 + 0.0818i	1551	0.0019 + 0.0095i	17397	0.0003 + 0.0011i
156	0.0201 + 0.0735i	1750	0.0017 + 0.0085i	19632	0.0003 + 0.0010i
176	0.0171 + 0.0661i	1975	0.0016 + 0.0076i	22154	0.0003 + 0.0009i
199	0.0146 + 0.0593i	2229	0.0014 + 0.0069i	25000	0.0002 + 0.0008i

Table 6.3 Admittances of the brushless DC motor in the d-axis

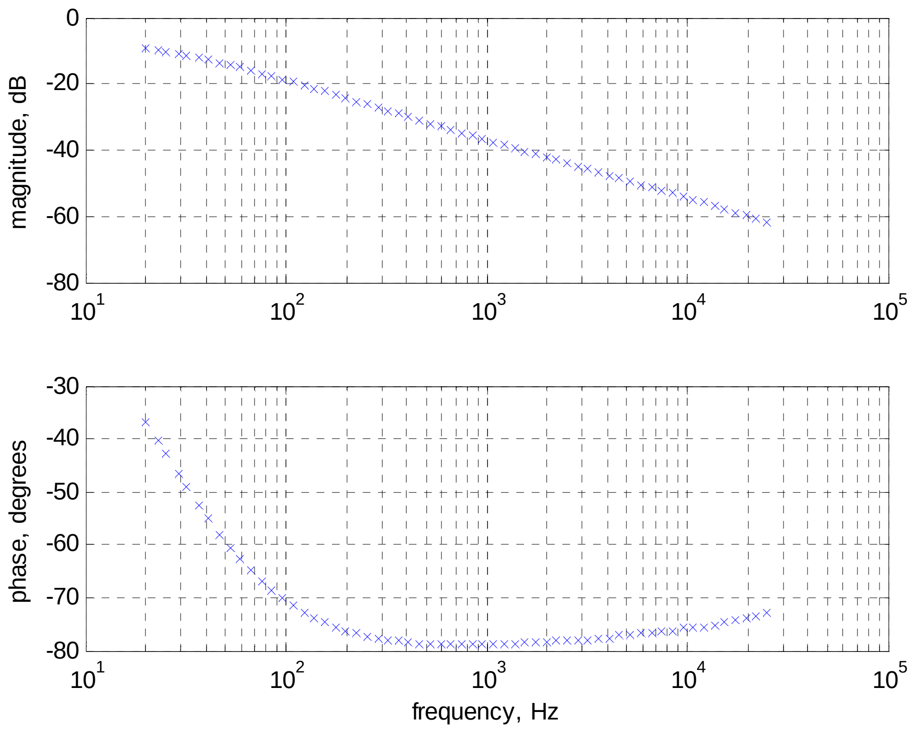


Figure 6.11 Magnitude and phase plot of admittance data

It is assumed that the transfer function of the admittance has the form

$$Y(s) = \frac{a_1}{\tau_1 s + 1} + \frac{a_2}{\tau_2 s + 1} + \cdots + \frac{a_n}{\tau_n s + 1},$$

where n is the order of the transfer function, a-s and τ-s are the parameters to be identified.

The fitness function F is defined as

$$F(a_1,\cdots,a_n,\tau_1,\cdots,\tau_n) = \frac{1}{\frac{1}{m}\sqrt{\sum_{k=1}^{m}\frac{|Y_k - Y(s_k)|}{|Y_k|}} + 10^{-12}},$$

where m is the number of admittance data set and $s_k = j2\pi f_k$.

In the example, the order of the transfer function to be estimated is assumed to be n = 6. The following is the fitness function m-file.

```
function fit=tfid_fit(parameters,data)
% tffit calculates fitness of transfer function fit
%
% t=tf_fit(parameters,data)
%
% Inputs:
% parameters = paramameter vector
%              (first gains, then time constants)
% data       = data.s - complex frequency vector
%              data.t - transfer function values
% Outputs:
% t       = vector of transfer function values
%           (same dimension as s)

o=(length(parameters))/2;

a   = parameters(1:o);
tau = parameters(o+1:2*o);

tpred = pftf(data.s,a,tau);

terror = data.t-tpred;

error = norm(terror./abs(data.t))/length(tpred);

fit = 1.0/(1.0e-12+error);
```

As can be seen, the first line of the fitness function determines the assumed order of the transfer function by inspection of the length of the parameter vectors. The parameter vector is then divided into the a and τ terms. The predicted transfer function is then calculated using the `pftf.m` function (which is included with the example files). The error and fitness are then calculated as previously described.

The mfile script which calls GOSET, `tfid.m`, is shown below. The first step is pre-processing the data and placing it into a data structure. The preprocessing is carried out by the script `ydata.m`, but we will not focus on this as it is discipline specific and not relevant to the use of GOSET.

```
% Transfer Function Identification Example
%
%   Stand Still Frequency Response (SSFR)
%   Impedance Data of Brushless DC Machine
%   Rotor is aligned with d-axis.

% Get admittance data
ydata;
data.s = s;
data.t = yd;

% Initialize the genetic algorithm parameters
GAP = gapdefault(1,1,200,400);  % Set default values for GAP
GAP.mc_alg = 6;
GAP.dt_alg = 3;

% Set range for genes
order = 6;
O=ones(1,order);
GAP.gd_min =  [ 1e-8*O 1e-8*O ];
GAP.gd_max =  [ 1e+1*O 1e+0*O ];
GAP.gd_type = [ 3*O 3*O ];
GAP.gd_cid  = [ 1*O 1*O ];

% Execute GOSET
[P,GAS, best] = gaoptimize(@tfid_fit,GAP,data);

% Plot the results
a   = best(1:order);
tau = best(order+1:2*order);
ydp = pftf(s,a,tau);
figure(2);
bodeplot(2,f,yd,'bx',ydp,'r-');
```

After the admittance data structure is assigned, the next step is establishing values for the genetic algorithm parameters stored in the GAP structure. As can be seen, the study was designed for a population size of 200 and 400 generations. Some of the GAP parameters were also set to values not equal to their defaults.

Next, GOSET is executed, and a comparison of the fitted transfer function to the data is given. The `bodeplot.m` function is used for this purpose (it is included with the example files). Executing the `tfid.m` script yields Figure 6.12.

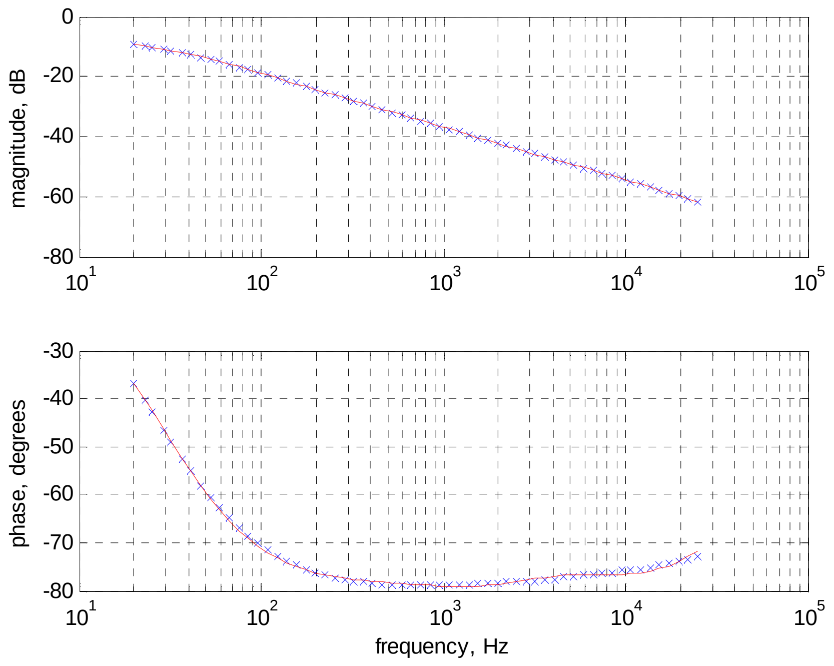


Figure 6.12 Plot of the magnitude and phase of the measured transfer function data and the transfer function obtained using GOSET

The magnitude and phase plot of the data set and the estimated transfer function are shown in Figure 6.12. The solid red line is for the transfer function estimated using GOSET and the blue x's are the measured transfer function values. The estimated transfer function fits the measured data very closely.

References

[CHO96] E. K. P. Chong and S. H. Żak, *An Introduction to optimization*, Wiley-Interscience, 1996

[TAN95] M. Tanaka, *GA-based decision support system for multi-criteria optimization*, Proceedings of the International Conference on Systems, Man and Cybernetics, Vol. 2, pp. 1556-1561.

6.5 Additional Problems

A large number of problems related to the design of inductors, electromagnetics, transformers, and rotating machinery are set forth and solved using GOSET in [SUD14]. Codes are included.

References

[SUD14] S.D. Sudhoff, *Power Magnetic Devices: A Multi-Objective Design Approach*, IEEE-Press/Wiley, 2014

Appendix

Appendix A. GOSET function list

Initialization	`gapdefault`	contains default parameter values for GAP
	`downsize`	reduce the population size to a desired number
	`gainit`	initialize the genetic algorithm
	`unrndinit`	initialize a population randomly
	`gasetup`	sets up a population of chromosomes
Genetic operators	`gaoptimize`	GOSET main routine
	`objwght`	generate weight vector for multi-objective functions
	`divcon`	prevent crowding of the chromosomes
	`scale`	determine the scaled fitness
	`select`	select chromosomes for reproduction
	`death`	determine parents to be replaced by children
	`matingcrossover`	exchanges genes between chromosomes
	`mutate`	randomly change some gene values
	`generepair`	fix gene value after crossover and mutation
	`migrate`	move chromosomes from one region to another
	`updateage`	update the age of all individuals
	`evaluate`	evaluate the fitness values of chromosomes
	`elitism`	preserve best chromosomes
	`randsearch`	search the vicinity of the best chromosomes
	`updatestat`	update the statistic information of GAS structure
	`normgene`	updates the normalized genes based on raw genes
	`rawgene`	updates the raw genes based on normalized genes
	`nondom`	find the non-dominated solutions
	`trimga`	perform a deterministic optimization
Plotting	`reportplot`	plots current population
	`distplot`	plots the distribution of the genes
	`paretoplot`	plots the population in the objective space
Misc.	`contents`	contains general information on GOSET

Appendix B. GOSET function reference

contents

Purpose Contain general information regarding GOSET

Syntax `contents`

Arguments None

Value None

Description `contents.m` has descriptions on data structures P, GAS and functions of GOSET and version information

See Also `gapdefault`

death

Purpose Determine parents that are replaced by the children

Syntax `Dlist = select(Pin,Plist,GAP)`

Arguments

`Pin`	structure of current population
`Plist`	parent list generated from select algorithm
`GAP`	structure of genetic algorithm parameters

Value `Dlist` death list

Description Death operator determines which individual is to die and replaced by the children. The followings are possible options for the death operators.

Replacing parents (`GAP.dt_alg = 1`)

Parents are replaced by their own children.

Random selection (`GAP.dt_alg = 2`)

The parents to be replaced are randomly chosen.

Tournament on fitness (`GAP.dt_alg = 3`)

The parent to be replaces is determined via the tournament based on the aggregate fitness value. `GAP.dt_nts` number of parents are randomly chosen for a tournament and the one with worst aggregate fitness value is marked for death.

Tournament on age (`GAP.dt_alg = 4`)

The parent to be replaces is determined via the tournament based on the age. Among the randomly chosen `GAP.dt_nts` number of parents, the oldest one is selected and marked for death.

Custom algorithm (`GAP.dt_alg = 5`)

User defined custom death algorithm is used. The handle of the custom function is assigned to `GAP.dt_cah`. The custom function must have the following format

```
D_list = f(region,size,age,mfit,fit)
```

`D_list`	indices of the individuals to be replaced by children
`region`	the region number
`size`	number of individuals for the death list

`age`	vector describing ages of the individuals in population
`mfit`	array with raw fitness values of the individuals in the region
`fit`	vector with aggregate fitness values of individuals in the region

As an example of a custom algorithm, the random death algorithm is written as an mfile called 'customdeath.m' which is shown below.

```
% Custom death algorithm example – random death algorithm
function dlist = customdeath(region,size,region_age,region_mfit,region_fit)

% Randomly select death list
regionsize=length(region_age);
randomlist = randperm(regionsize);

dlist = randomlist(1:size);
```

This mfile must exist in the same folder as the fitness function file or in the GOSET Core folder. Then the custom file handle `GAP.dt_cah` is set to `@customdeath`.

Random algorithm (`GAP.dt_alg = 6`)

If this option is selected, the death algorithm is randomly chosen among the first four death algorithms at each generation.

despick

Purpose To select a particular solution from the pareto-optimal front for 2 objectives.

Syntax

```
[geneval,fit]     = despick(bI,f)
[geneval]         = despick(bI,f)
```

Arguments

`bI`	Best individuals (non-dominated solutions)
`f`	fitness values
`geneval`	value of the genes of the selected non-dominated solution
`fit`	fitness of the selected non-dominated solution

Value None

Description A function to pick and obtain fitness and gene values of a particular non-dominated solution from the Pareto Optimal front. It also plots the Pareto-Optimal front with the chosen solution highlighted. This works only with 2 objective optimization.

distplot

Purpose	Plot the distribution of the genes in the individuals
Syntax	`distplot(fignum,P,objective,GAP,[region])`

Arguments

`fignum`	figure number
`P`	structure of current population
`objective`	objective function number to show in the plot
`GAP`	structure of genetic algorithm parameters
`region`	plot only the individuals in this specified region (optional)

Value None

Description `distplot` shows the distribution of the genes of the individuals. It is called within the `reportplot` and plotted together with the fitness history.

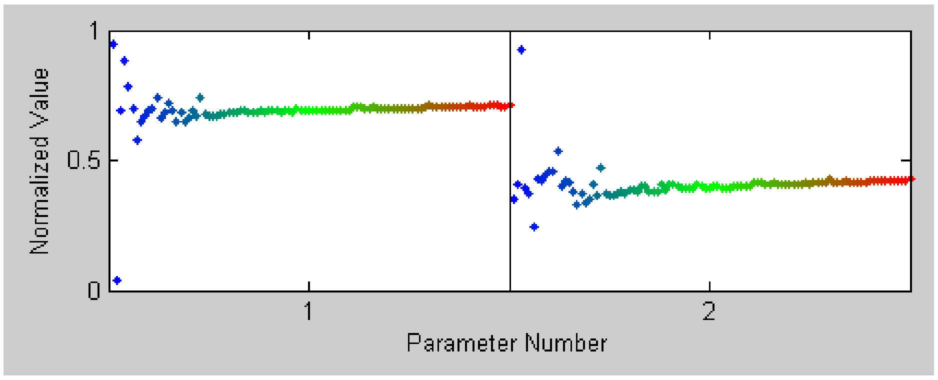


Figure B.1

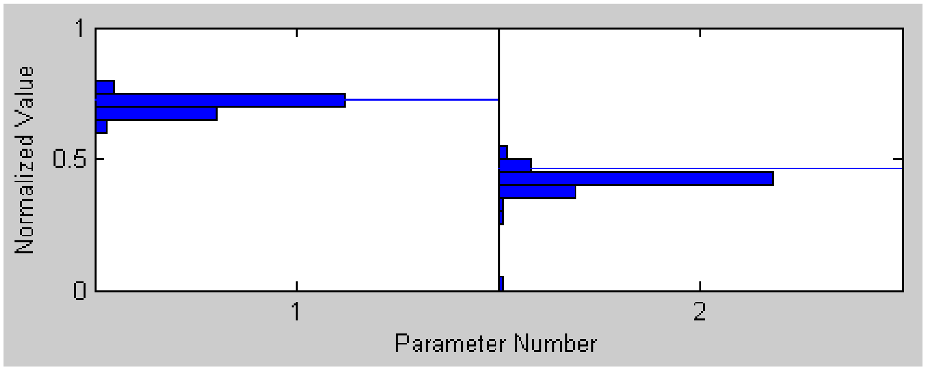


Figure B.2

There are two types of distribution plot. Setting `GAP.dp_type` to 1 will show the first type of distribution plot which displays the normalized gene values as in Figure B.1. In this case, genes of the least fit individuals are plotted towards the left of the window blue; genes of an average fit individual are plotted towards the center of the window; and genes of the most fit individuals are plotted towards the right of the window in red.

The second type (`GAP.dp_type = 2`) of distribution plot shows the histogram of the normalized gene values as in Figure B.2. The number of bars for the histogram can be set using `GAP.dp_res`. In Figure B.2, the number of bars is set to `5` (`GAP.dp_res = 5`). The gene values of the best individual of each region are indicated by green horizontal lines. For each gene values, there are as many green lines as the number of regions.

For both of the distribution plot, only a part of the population can be displayed by setting the parameter `GAP.dp_np` that determines the maximum number of individuals to plot. Only `GAP.dp_np` individuals are randomly chosen from the population and displayed. The positions of green lines represent the normalized gene values of the best individual.

See also `reportplot, paretoplot`

divcon

Purpose Compute penalty function values for maintaining diversities of the population

Syntax `penalty = divcon(Pin,GAP)`

Arguments

`Pin`	structure of current population
`GAP`	structure of genetic algorithm parameters

Value

`penalty`	penalty function vector

Description Maintaining genetic diversity in the population is important especially in the multi-objective optimization problem. Diversity control algorithms are employed so that the under represented individuals are emphasized and similar individuals are penalized by degrading their fitness values.

Diversity control can be applied to either the parameter (solution) space or the fitness function space. Setting `GAP.dc_spc = 1` causes the diversity control in the parameter space and setting `GAP.dc_spc = 2` causes the diversity control in the fitness function space.

Presently, four different diversity control algorithms are used in GOSET.

Diversity control algorithm 1

This algorithm is chosen by setting `GAP.dc_alg = 1`. For each individual, the distances with all other individuals are evaluated. Then the number of individuals, whose distance from the individual of interest is smaller than the threshold distance, is counted. The threshold distance is randomly determined as a value between the minimum threshold (`GAP.dc_mnt`) and the maximum threshold (`GAP.dc_mxt`). That is,

Threshold distance = average distance among the individual H α

where α =(`GAP.dc_mnt`+randH(`GAP.dc_mxt-GAP.dc_mnt`)). Then the penalty function value of an individual is defined as the reciprocal of the counted number of individuals.

Diversity control algorithm 2

This algorithm is chosen by setting `GAP.dc_alg = 2`. To overcome the problem of the computational load in the first method, this algorithm uses a weighted sum of gene values for diversity control. For an arbitrary weight vector whose element number is same as gene number in an individual, the weighted sum of each individual is evaluated. Then the modulus after dividing the weighted sum by 1 is taken. If the gene values of individuals are very similar, then the modulus of the weighted sum must be also similar. Then the individuals are grouped according to the modulus values and put into a corresponding bin. The number of bins, that is the number of groups, is randomly determined as the following

$$\text{No. of bins} = \text{round}\,(\alpha \cdot \text{Number of individual}),$$

where α = `GAP.dc_mnb+rand(GAP.dc_mxb-GAP.dc_mnb)`. Then the interval [0,1] is divided into (No. of bins) equally distanced subintervals. The penalty value of an individual is the reciprocal of the total number of individuals in the same bin.

However, even with different gene values, individuals may have similar modulus for some weight vectors. In such cases, the penalty value does not reflect the actual proximity of gene values. Hence, the procedure is repeated `GAP.dc_ntr` times and the largest penalty function value is chosen as the final penalty function value for each individual.

Diversity control algorithm 3

This algorithm is chosen by setting `GAP.dc_alg` = 3. The idea of this diversity control algorithm is similar to the diversity algorithm 1. The sum of infinity norm between the solutions is used to determine the penalty value as shown in the following formula

$$P_{pen}^{k} = \frac{1}{\sum_{i=1} \exp\left(-\frac{d_{i,k}}{d_c}\right)},$$

where $d_{i,k}$ is the infinity norm between k'th and i'th individual and d_c is the distance constant (`GAP.dc_dc`) which controls the size of the neighborhood. As the distance constant d_c increases, the effective size of the neighborhood increases and the penalty level also increases.

Diversity control algorithm 4

This algorithm is chosen by setting `GAP.dc_alg` = 4. It is identical to the diversity control algorithm 3 except the fact that only a part of the population, that is, for each individual, `GAP.dc_nt` individuals are randomly chosen and used in the distance evaluation. The following formula is used to calculate the fitness penalty weight for *k*'th individual.

$$P_{pen}^{k} = \frac{1}{1 + \frac{\text{Number of population in the region}}{\texttt{GAP.dc\_nt}} \sum_{i=1}^{\texttt{GAP.dc\_nt}} \exp\left(-\frac{d_{i,k}}{d_c}\right)}$$

where $d_{i,k}$ is the infinity norm between *k*'th and *i*'th individual, d_c is the distance constant (`GAP.dc_dc`)

downsize

Purpose Reduce the population to a desired size

Syntax `M = downsize(P,newsize)`

Arguments

`P`	structure of current population
`newsize`	the size of the new population

Value

M	structure of downsized population

Description This function reduces the size of the population to a desired number based on the (cumulative) rank of the individual.

In the multiple region (multi-population) case, the number of individuals in a region is determined such that the ratio of individuals among regions is maintained. For example, suppose a population with 100 individuals that are distributed in 3 different regions as in the following table. If we want the new population to have only 50 individuals, then the number of individuals in the new populations becomes the half of the number of individuals in the original population as shown in Table B.1.

Region	1	2	3	total
No. of individuals in P	30	50	20	100
No. of individuals in M_t	**15**	**25**	**10**	**50**

Table B.1

The selection of the individuals is based on the rank in single objective case. In the multi-objective case, cumulative rank is used to pick the individuals for the new population. Consider a 3-objective optimization problem in Table B.2. If we have four individuals and need to reduce the size to two, then individual **A** and **D** are selected according to the cumulative rank.

Individual	Rank in each objective			Cumulative Rank
	I	II	III	
A	3	1	1	**5**
B	4	4	2	10
C	2	3	4	9
D	1	2	3	**6**

Table B.2

elitism

Purpose Preserve the best individuals

Syntax `O = elitism(N1,N0,GAP,GAS)`

Arguments

`N1`	structure of current manipulated population
`N0`	structure of original population
`GAP`	structure of genetic algorithm parameters
`GAS`	structure of genetic algorithm statistics

Value

`O`	structure of output population

Description Elitism is activated by setting `GAP.el_act` = 1. The starting point of elitism can be using the parameter `GAP.el_fgs` that specifies the fraction of the population. For example, if `GAP.el_fgs` = 0.25 with the total generation number of 100, then the elitism is effective starting from 25th generation.

In single objective optimization problems, the best individual in each region of the processed population and the best one in the original population are compared. If the best individual of the processed population is worse than that of the original population, then the best one in the processed population is replaced by the best one in the old population. In multi-objective optimization problem, it is guaranteed that a limited number of non-dominated individuals of the population are preserved up to certain number. The maximum number of preserved non-dominated individuals is determined by (population size × `GAP.el_fpe`).

The elite fraction `GAP.el_fpe` is dynamically adjusted within the initial value `GAP.el_fpe0` and final value `GAP.el_fpef` defined by the user. That is, the fraction of elite `GAP.el_fpe` at each generation is calculated as

$$\texttt{GAP.el\_fpe} = \texttt{GAP.el\_fpe0} \times (1-\beta) + \texttt{GAP.el\_fpef} \times \beta$$

where $\beta = \frac{\text{current generation number}}{\text{total number of generation}}$. Thus the elite ratio in the population starts from the initial value `GAP.el_fpe0` and varies gradually to the final value `GAP.el_fpef`.

evaluate

Purpose	Evaluate the fitness of chromosomes
Syntax	`[mfit,es,une] = evaluate(P,GAP,cne,D)`

Arguments

`P`	structure of current population
`GAP`	structure of genetic algorithm parameters
`cne`	current number of evaluations performed
`D`	an optional data structure used for fitness evaluation

Value

`mfit`	multi-objective fitness
`es`	evaluation status of each member of population
`une`	updated number of evaluations

Description `evaluate` assigns individuals with fitness values obtained from the fitness function defined by `P.fithandle`.

With GOSET 2.6, a new feature is the ability to utilize parallel processing, so that multiple cores are used to calculate the fitness of the population. Note that this requires the use of MATLAB’s Parallel Processing Toolbox. Setting `GAP.ev_pp = true` invokes this option. The number of evaluation groups is set to `GAP.ev_npg` which has a default value of 12, and should be set to the number of cores being used.

When `GAP.ev_are` is set to 0, this function only updates the individuals whose fitness values have not been evaluated. When `GAP.ev_are = 1`, the fitness value of all the individuals are evaluated.

Also `GAP.ev_bev` determines whether to pass all the individuals to the fitness evaluation function at the same time (when set to 1) or to evaluate one individual at a time (when set to 0). The fitness function must be written to handle the vector evaluation.

Normally, the only gene values are passed to the fitness function. If the supplementary data flag `GAP.ev_ssd = 1`, then the age (`P.age`), previous fitness values (`P.mfit`) and the region (`P.region`) are also sent to the fitness function.

`D` is the optional data structure that is required for evaluating the fitness function and it is passed to the fitness function if it is defined when the `gaoptimize` is called.

The passed data and its order are listed in the following table.

Optional data `D`	`GAP.ev_ssd`	Data and its order passed to the fitness function
exists	0	`P.gene, D`
	1	`P.gene, P.age, P.mfit, P.region, D`
does not exist	0	`P.gene`
	1	`P.gene, P.age, P.mfit, P.region`

gainit

Purpose Initialize the genetic algorithm

Syntax `[GAP,GAS,Pk]=gainit(fitfun,D,GAP,GAS,iP)`

Arguments

`@fitfun`	name of the m-file that evaluates the fitness
`D`	optional data needed by fitness function
`GAP`	structure of genetic algorithm parameters
`GAS`	structure of genetic algorithm statistics
`iP`	optional initial population

Value

`GAP`	structure of genetic algorithm parameters
`GAS`	structure of genetic algorithm statistics
`Pk`	structure of the population

Description `gainit` initializes the genetic algorithm by setting up the population. If the optional initial population is passed, `gainit` only evaluates the fitness of the population. Otherwise `gasetup` is called to generate initial population, and the fitness is evaluated. If the size of the initial population (`GAP.fp_ipop`) is larger than the steady state population (`GAP.fp_ipop`), the population size is reduced. In the last step, `gainit` generates a report on initial evaluation.

See Also `gasetup`

gaoptimize

Purpose	Perform function optimization using GOSET

Syntax

```
[fP,GAS]     =gaoptimize(@fitfun,GAP,D,GAS,iP)
[fP,GAS]     =gaoptimize(@fitfun,GAP,D)
[fP,GAS]     =gaoptimize(@fitfun,GAP)
[fP,GAS,bI]  =gaoptimize(@fitfun,GAP,D,GAS,iP)
[fP,GAS,bI]  =gaoptimize(@fitfun,GAP,D)
[fP,GAS,bI]  =gaoptimize(@fitfun,GAP)
[fP,GAS,bI,f]=gaoptimize(@fitfun,GAP,D,GAS,iP)
[fP,GAS,bI,f]=gaoptimize(@fitfun,GAP,D)
[fP,GAS,bI,f]=gaoptimize(@fitfun,GAP)
```

Arguments

`@fitfun`	name of the m-file that evaluates the fitness
`GAP`	structure of genetic algorithm parameters
`D`	optional data required by fitness function
`GAS`	structure of genetic algorithm statistics
`iP`	optional variable with initial population

Value

`fP`	structure of final population
`GAS`	structure of genetic algorithm statistics
`bI`	best individual (gene values for the best individual)
`f`	best fitness/fitnesses (fitness values of individuals in `bI`)

Description As the main function of GOSET, it performs the function optimization using GOSET. The structure of `gaoptimize.m` is modularized. Thus users who want to experiment their own operator, can easily modify this function.

gapadjust

Purpose Sets the mutation parameters based on generation number

Syntax [GAP] = gapadjust(GAS,GAPic)

Arguments

`GAS`	structure of genetic algorithm statistics
`GAPic`	structure of initial genetic algorithm parameters

Value `GAP` structure of genetic algorithm parameters

Description `gapadjust` is called within `gaoptimize` in order to set the mutation and elitism parameters for that generation. Note that `GAPic` is normally simply taken to be `GAP`.

See Also `gaoptimize, mutate`

gapdefault

Purpose Assigns default values to the genetic algorithm parameters used in GAP

Syntax

```
GAP = gapdefault
GAP = gapdefault(nobj)
GAP = gapdefault(nobj,obj,npop,ngen)
```

Arguments

`nobj`	number of objectives
`obj`	objective to optimize (default is 0 (multiobjective) if nobj>1)
`npop`	nominal population size (default is 100)
`ngen`	number of generations (default is 100)

Value

`GAP`	structure of genetic algorithm parameters

Description This function returns the structure of genetic algorithm parameters GAP with their default values. The user can load the `gapdefault` and then redefine only the required fields, instead of defining all the fields.

The default values defined in `gapdefault` are list in the Appendix C.

gasetup

Purpose Set up a population of chromosomes

Syntax `[P,GAS] = gasetup(popsize,GAP,@fitfun,[D])`

Arguments

`popsize`	number of individuals in the population
`GAP`	structure of genetic algorithm parameters
`@fitfun`	name of the m-file that evaluates the fitness
`D`	optional data needed by fitness function

Value

`P`	structure of the population
`GAS`	structure of genetic algorithm statistics

Description `gasetup` is called within `gainit` when the initial population does not exist.

See Also `gaoptimize, mutate`

generepair

Purpose	correct the gene value to be feasible	
Syntax	`[rgene] = generepair(gene,GAP)`	
Arguments	`gene`	an individual, vector of normalized gene values
	`GAP`	structure of genetic algorithm parameters
Value	`rgene`	repaired gene values

Description `generepair` is called within `matingcrossover` and `mutate` to correct any resultant genes which lie outside the specified range. The parameter `GAP.gr_alg` controls the repair method.

By default, `GAP.gr_alg` is set to 1 for *hard limiting* method that clips any illegal gene value to the boundary value. For example, if a resultant gene value is 1.2, it is adjusted to 1, and if it is -0.4, it is adjusted to 0.

By setting it to 2, *ring mapping* method is applied and the modulus after division by 1 is used as the repaired value. For example, if a resultant gene value is 1.2, it is adjusted to 0.2, and if it is -0.1, it is adjusted to 0.9.

In situations where the limit of a variable is a physical limit which also happens to be the location of the optimum solution, the hard limiting method results in significantly better performance.

matingcrossover

Purpose Perform mating and genetic crossover on a population

Syntax

```
N = matingcrossover(P,P_list,PL_size,
                             Dlist,GAP,GAS)
```

Arguments

`P`	structure of current population
`P_list`	parent list from selection operator
`PL_size`	size of the parent list
`Dlist`	death list from death operator
`GAP`	structure of genetic algorithm parameters
`GAS`	structure of genetic algorithm statistics

Value

N	structure of the population after crossover

Description Perform crossover operations on a population. Three different types of crossover methods are used in GOSET; single point crossover, simple blend crossover, and simulated binary crossover.

The parameter `GAP.mc_pp` specifies the mating crossover probability, that is, the fraction of the population replaced by children. The fraction of the chromosome undergoes crossover is determined by `GAP.mc_fc`.

All crossover operation is region specific and parents that are selected from one region reproduce children into the same region. Also all the crossover operations are chromosome-ID specific. Hence genes of different chromosome ID are treated separately and the crossover operators are applied independently.

The mating crossover methods are determined by `GAP.mc_alg` as in the following table.

`GAP.mc_alg`	**Mating Crossover method**
1	Single point crossover
2	Scalar simple blend crossover
3	Vector simple blend crossover
4	Scalar simulated binary crossover
5	Vector simulated binary crossover
6	Random algorithm

Let’s discuss the mating crossover methods one by one.

Single point crossover

This crossover operator is similar to the crossover operator in binary-coded GAs. A crossover point is randomly selected and the gene values after that point are swapped between two parent chromosomes.

If P_1 and P_2 are the parent chromosomes with n genes and c is the crossover point, then the children chromosomes are

$$C_1 = [P_{1;(1:c-1)} \;\; P_{2;(c:n)}] \text{ and } C_2 = [P_{2;(1:c-1)} \;\; P_{1;(c:n)}]$$

where $P_{1;(a:b)}$ is a vector whose elements are gene values from a'th to b'th positions of P_1.

Scalar simple blend crossover

Scalar simple blend crossover generates the children from the weighted sum of their parents by the following steps;

STEP 1 : For i'th gene, choose a random number $u_i \in [-1,1]$

STEP 2 : Calculate the average of the parents

$$m_i = \frac{P_{1;i} + P_{2;i}}{2}$$

STEP 3 : Calculate the amount of change

$$\delta_i = u_i \cdot \left| P_{1;i} - P_{2;i} \right|$$

STEP 4 : Compute the offspring

$$C_{1;i} = m_i + \delta_i \text{ and } C_{2;i} = m_i - \delta_i$$

Note that each gene in the same chromosome is crossovered with the different amount of change.

Vector simple blend crossover

Vector simple blend crossover is similar to the scalar simple blend crossover. The only difference is that all genes in the same chromosome are crossovered with the same amount of change as in the following steps

STEP 1 : Choose a random number $u \in [-1,1]$

STEP 2 : Calculate the average of the parents

$$m_i = \frac{P_{1;i} + P_{2;i}}{2}$$

STEP 3 : Calculate the amount of change

$$\delta_i = u \cdot \left| P_{1;i} - P_{2;i} \right|$$

STEP 4 : Compute the offspring

$$C_{1;i} = m_i + \delta_i \text{ and } C_{2;i} = m_i - \delta_i$$

Note that all genes in the same chromosome are crossovered with the same amount of change.

Scalar simulated binary crossover

Scalar simulated binary crossover generates the children by the following steps;

STEP 1 : For *i*'th gene, choose a random number $u_i \in [0,1]$

STEP 2 : Calculate the spread factor

$$\beta_i = \begin{cases} (2u_i)^{\frac{1}{\eta_c+1}}, & \text{if } u_i \le 0.5; \\ \left(\frac{1}{2(1-u_i)}\right)^{\frac{1}{\eta_c+1}}, & \text{otherwise.} \end{cases}$$

where η_c is the distribution tightness parameter `GAP.mc_ec`.

STEP 3 : Compute the offspring

$$\begin{aligned} C_{1;i} &= 0.5[(1+\beta_i)C_{1;i} + (1-\beta_i)C_{2;i}], \\ C_{2;i} &= 0.5[(1-\beta_i)C_{1;i} + (1+\beta_i)C_{2;i}]. \end{aligned}$$

Note that each gene in the same chromosome can be recombined with different spread factor.

Vector simulated binary crossover

Vector simulated binary crossover is identical as scalar simulated binary crossover except that the spread factor is same for all the genes in the same chromosome.

The following describes the vector simulated crossover;

STEP 1 : Choose a random number $u \in [0,1]$

STEP 2 : Calculate the spread factor beta

$$\beta = \begin{cases} (2u)^{\frac{1}{\eta_c+1}}, & \text{if } u \le 0.5; \\ \left(\dfrac{1}{2(1-u)}\right)^{\frac{1}{\eta_c+1}}, & \text{otherwise.} \end{cases}$$

STEP 3 : Compute the offspring

$$\begin{aligned} C_1 &= 0.5[(1+\beta)C_1 + (1-\beta)C_2], \\ C_2 &= 0.5[(1-\beta)C_1 + (1+\beta)C_2]. \end{aligned}$$

Random crossover

For every `GAP.mc_gac` generation, a mating crossover methods are randomly selected from the five mating crossover methods described above.

See Also `generepair`

Reference K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, Chichester, UK, 2001

migrate

Purpose Change the region of individuals

Syntax `O = migrate(N,GAP,cg)`

Arguments

`N`	structure of population before migration
`GAP`	structure of genetic algorithm parameters
`cg`	current generation number

Value

`O`	structure of population after migration

Description This function works only when there are multiple regions. If the migration occurs, some individuals are selected and moved to other regions. The migration interval is randomly chosen from the integer values between 0.5×`GAP.tmig` and 1.5×`GAP.tmig.` For example, if `GAP.tmig = 6` then, the migration interval can be any integer from 3 to 9. Each individual is selected and migrated with the probability of `GAP.pmig`. The target region is chosen randomly among `GAP.nreg` number of regions.

mutate

Purpose Perform mutation on a population of chromosomes

Syntax `O = mutate(N,GAP)`

Arguments		
	`N`	structure of population before mutation
	`GAP`	structure of genetic algorithm parameters

Value `O` structure of the population after mutation

Description This function applies genetic mutation on the population. Four different mutation algorithms are applied sequentially in the order of total mutation, partial mutation, vector mutation, and integer mutation.

These mutation operations are performed on the normalized gene values. When a gene value lies outside of the allowed range after mutation, then its value is corrected using `generepair` routine.

Total mutation

Each gene can be mutated to any value within the predetermined range with the probability of `GAP.mt_ptgm`. Thus, the mutated genes have no relationship to their previous value.

Partial mutation

Each gene can be perturbed with respect to its current value by using a random value generated using a Gaussian random variable. The mutated gene value is related to the original gene value.

Relative gene mutation

In the relative gene perturbation, with the probability of `GAP.mt_prgm`, each gene value is perturbed by certain fraction of the current gene value. The amount of perturbation is determined using a Gaussian random variable with standard deviation of `GAP.mt_srgm`.

The relative gene mutation on *j*'th gene in *k*'th individual can be expressed as

$$P_{ng;j,k} = P_{ng;j,k} \cdot (1 + N(0, \sigma_{rvm}))$$

where $N(0,\sigma_{rvm})$ is a Gaussian random variable with mean 0 and standard deviation σ_{rvm}(`GAP.mt_srgm`).

Absolute gene mutation

In the absolute gene perturbation, each gene value is added with a Gaussian random variable with standard deviation of `GAP.mt_sagm`. The probability of absolute gene perturbation is defined in `GAP.mt_pagm`.

The absolute gene mutation on *j*'th gene in *k*'th individual can be expressed as

$$P_{ng;j,k} = P_{ng;j,k} \cdot (1 + N(0,\sigma_{rvm}))$$

where $N(0,\sigma_{rvm})$ is a Gaussian random variable with mean 0 and standard deviation σ_{rvm}(`GAP.mt_srgm`).

Vector mutation

This function is similar to partial mutation except the fact that all the genes of an individual are involved.

Relative vector mutation

Each individual undergoes the relative vector mutation with the probability of `GAP.mt_prvm`. Every gene value of the individual is perturbed by certain fraction of the current gene value. The relative vector mutation on the *k*'th individual can be expressed as

$$P_{ng;k} = P_{ng;k} \cdot (1 + v_{dir} \cdot N(0,\sigma_{rvm}))$$

where v_{dir} is a normalized random vector $(P_{ngenes} \times 1)$ specifying the direction of perturbation and $N(0,\sigma_{rvm})$ is a Gaussian random variable with mean 0 and standard deviation σ_{rvm}(`GAP.mt_srvm`).

Absolute vector mutation

Each individual undergoes absolute vector mutation with the probability of `GAP.mt_pavm`. The absolute vector mutation on the *k*'th individual can be expressed as

$$P_{ng;k} = P_{ng;k} + v_{dir} \cdot N(0, \sigma_{avm})$$

where v_{dir} is a normalized random vector ($P_{ngenes} \times 1$) specifying the direction of perturbation and $N(0, \sigma_{avm})$ is a Gaussian random variable with mean 0 and standard deviation σ_{avm} (`GAP.mt_savm`).

Integer mutation

Each integer gene can be mutated to any integer value within the predetermined range with the probability of `GAP.mt_pigm`.

Each mutation related parameters is dynamically updated within the initial value and final value defined by the user. For example, the total mutation probability `GAP.mt_ptgm` at each generation is calculated as

GAP.mt_ptgm = GAPic.mt_ptgm0 $\times (1-\beta)$ + GAPic.mt_ptgmf $\times \beta$

where $\beta = \frac{\text{current generation number}}{\text{total number of generation}}$. Thus the total mutatio probability starts from the initial value `GAP.mt_ptgm0` and varies gradually to the final value `GAP.mt_ptgmf`.

See Also `generepair`

nondom

Purpose Find the set of non-dominated solutions for multi-objective optimization

Syntax `nd = nondom(f,t)`

Arguments

`f`	a matrix of objective function values whose dimension is (Number of objective functions) by (Number of solutions)
`t`	1 indicates that the larger objective value is better 0 indicates that the smaller objective value is better

Value

nd	a row vector with dimension equal to the number of solutions whose elements are 1 if the solutions are non-dominated and 0 if they are dominated

Description `nondom` is used to identify the non-dominated solutions among the solutions using the objective function value matrix. The method proposed by Kung et al. is employed.

Kung et al.'s method of identifying the non-dominated solution set

Step 1 Sort the population according to the descending order of importance in the first objective function and name the population as P

Step 2 **Front**(P)

```
IF   |P| = 1,
          Return P as the output of Front(P)
ELSE
        T = Front ( P(1: [ |P|/2 ]) )
        B = Front ( P( [ |P|/2 – 1 ] : |P|) )

        IF   the i-th non-dominated solution of B is not dominated by
             any non-nominated solution of T,
               M=T ∪{i}
               Return M as the output of Front(P)
END
```

Note 1. |•| is the number of the elements
2. P(a : b) means all the elements of P from index a to b,
3. [•] is an operator gives the nearest smaller integer value.

It is a recursive algorithm, and it may not be easy to visualize. However, it is the most computationally efficient method known at the time this manual is written.

Examples Suppose we have the following objective function value matrix with two objectives and five solutions ,

$$O = \begin{bmatrix} 4 & 6 & 9 & 8 & 2 \\ 5 & 1 & 6 & 7 & 4 \end{bmatrix}.$$

Then `Nd = nondom(O,1)` returns

Nd = [0 0 1 1 0].

Reference K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, Chichester, UK, 2001, pp. 38-39

normgene

Purpose Update the normalized gene values based on the raw gene values

Syntax `P = normgene(P)`

Arguments `P` structure of population before updating the normalized gene values

Value `P` structure of population after updating the normalized gene values

Description `normgene` updates the normalized gene values (`P.normgne`) based on the actual gene values (`P.gene`). The raw gene value is mapped to a value between 0 and 1 according to the type of the gene (`P.type`). Note that only the population members who have not been evaluated are updated.

The following table shows how `normgene` maps the raw gene value to the normalized gene values on *j*'th gene of the *m*'th chromosome for different types of gene.

Gene type	P_{type}	Operation
Integer & linear	1, 2	$P_{\text{ng};j,k} = (P_{\text{g};j,k} - P_{\text{min};j})/(P_{\text{max};j} - P_{\text{min};j})$
Logarithmic	3	$P_{\text{ng};j,k} = \dfrac{\ln(P_{\text{g};j,k}) - \ln(P_{\text{min};j})}{\ln(P_{\text{max};j}) - \ln(P_{\text{min};j})}$

Table B.3

Examples With the following parameters

$$P_{\text{min}} = [\ 0\ \ 1\ \ 10\ ],\ P_{\text{max}} = [\ 10\ \ 2\ \ 1000\ ],\ \text{and}\ P_{\text{type}} = [\ 1\ \ 2\ \ 3\ ],$$

if a chromosome with normalized gene values is

$$P_{g;k} = [\ 5\ \ 1.5\ \ 500\ ],$$

then the corresponding chromosome with actual gene values is

$$P_{ng;k} = [\ 0.5\ \ 0.5\ \ 0.8495\ ].$$

See Also `rawgene`

objwght

Purpose Create an objective weight vector for use in multi-objective optimization

Syntax `owv = objwght(GAP)`

Arguments `GAP` structure of genetic algorithm parameters

Value `owv` normalized weight vector for scalarization of the multi-objective function values

Description `objwght` generates a normalized weight vector to be used for scalarization of the fitness function values in the multi-objective optimization problem.

In the single-objective optimization problem where `GAP.fp_nobj = 1`, there is only one objective function. Thus `objwght` returns `owv = 1`

Even in the multi-objective optimization problem (`GAP.fp_nobj > 1`), it is possible to use one objective function value for fitness evaluation. The objective function number to be used is specified in `GAP.fp_obj`. Then the output weight vector `owv` has all zero values except for the element corresponding to the objective function specified by `GAP.fp_obj`.

Example Consider a multi-objective optimization with three objectives f_1, f_2 and f_3. A possible weight vector is

```
owv = [ 0.2  0.7  0.1 ].
```

Then the fitness value is calculated as

$$\text{Fitness} = 0.2\,f_1 + 0.7\,f_2 + 0.1\,f_3.$$

If `GAP.fp_obj = 2`, then `objwght` generates

```
owv = [ 0  1  0 ].
```

Hence the fitness value is calculated as

$$\text{Fitness} = f_2.$$

paretoplot

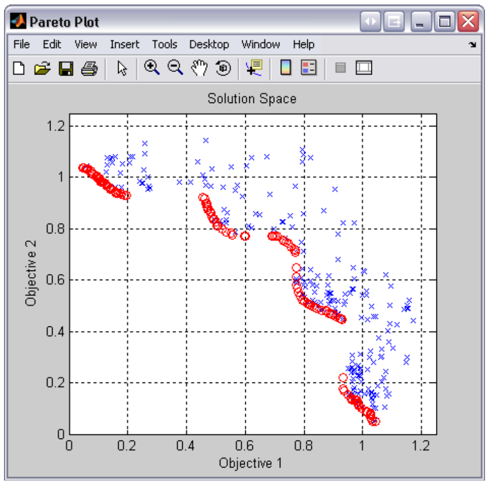
Purpose Plot two objective functions in 2D objective space

Syntax `paretoplot(P,GAP,[region])`

Arguments

`P`	structure of current population
`GAP`	structure of genetic algorithm parameters
`region`	an optional integer argument specifies the region of which the chromosomes are plotted

Description `paretoplot` generates 2D plot of 2 objective functions or 2D plot of 3 objective functions as in Figure B.4. It is called within `reportplot`.



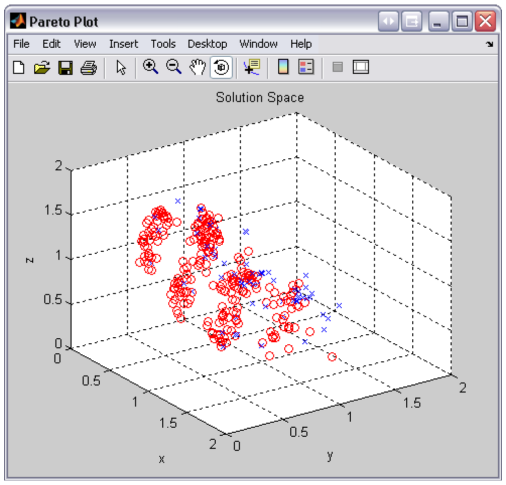


Figure B.4 2D and 3D Pareto plots

When the view angle is adjusted for the better observation in the case of 3D plot, it is maintained throughout the evolution process.

See also `reportplot, distplot`

randsearch

Purpose	Perform a random search in the vicinity of the best individual in each region for better individual
Syntax	[O,GAS] = `randsearch(M,GAP,GAS,D)`

Arguments

`M`	structure of current population
`GAP`	structure of genetic algorithm parameters
`GAS`	structure of genetic algorithm statistics
`D`	an optional data structure if needed for fitness evaluation

Value

`O`	structure of the population after the random search
`GAS`	structure of genetic algorithm statistics

Description `randsearch` explores the neighborhood of the best individual for better solution by random mutation of the best individual. By extensively exploring the vicinity of the best individual, it helps the GA to converge to the optimal solution faster.

There are two different random search operations. They are the relative random search that uses the relative vector mutation and the absolute random search that employs the absolute vector mutation. At each generation, only one of the two random search operations is active.

Random search starts at (`GAP.rs_fgs` × `GAP.fp_ngen`)'th generation and (`GAP.rs_fps` × `GAP.fp_npop`) individuals are randomly generated using relative vector mutation with the standard deviation of `GAP.rs_srp` or absolute vector mutation with the standard deviation of `GAP.rs_sap`. The choice between the two random mutations is dependant on the value `GAP.rs_frp`. `GAP.rs_frp` is the probability that the absolute mutation is used and thus the probability that the relative mutation is utilized is `(1-GAP.rs_frp)`.

After generating the mutants, the fitness values of the mutants are evaluated. If there exists an individual whose fitness is better than that of the current best individual, then the current best is replaced by the new individual.

rawgene

Purpose Update the raw gene values based on the normalized gene values

Syntax `P = rawgene(P)`

Arguments `P` structure of population before updating raw gene values

Value `P` updated structure of population after updating raw gene values

Description `rawgene` updates the actual gene values (`P.gene`) based on the normalized gene values (`P.normgene`). The normalized gene value is mapped to a value in the predefined range according to the type of the gene (`P.type`). Note that only the population members who have not been evaluated are updated.

The following table shows how `rawgene` maps the normalized gene value to the actual gene values on *j*'th gene of the *k*'th chromosome for different types of gene.

Gene type	P_{type}	Operation
Integer	1	$P_{g;j,k} = [(P_{max;j} - P_{min;j}) \cdot P_{ng;j,k} + P_{min;j}]$ where [ ] is the round -up operator
Real	2	$P_{g;j,k} = (P_{max;j} - P_{min;j}) \cdot P_{ng;j,k} + P_{min;j}$
Logarithmic	3	$P_{g;j,k} = \exp\left(\left(\ln(P_{max;j}) - \ln(P_{min;j})\right) \cdot P_{ng;j,k} + \ln(P_{min;j})\right)$

Table B.4

Examples With the following parameters

$$P_{min} = [\ 0\ \ 1\ \ 10\ ],\ P_{max} = [\ 10\ \ 2\ \ 1000\ ],\ \text{and } P_{type} = [\ 1\ \ 2\ \ 3\ ],$$

if a chromosome with normalized gene values is

$$P_{ng;k} = [\ 0.5\ \ \ 0.5\ \ \ 0.5\ ],$$

then the corresponding chromosome with actual gene values is

$$P_{g;k} = [\ 5\ \ \ 1.5\ \ 100\ ].$$

See Also `normgene`

reportplot

Purpose Plot the distribution of the genes of the chromosomes, the fitness history and Pareto plot

Syntax `reportplot(GAP,GAS,Pk)`

Arguments

`GAP`	structure of genetic algorithm parameters
`GAS`	structure of genetic algorithm statistics
`P`	structure of the current population

Value None

Description Plots the distribution of the genes of the chromosomes with the fitness history as in Figure B.5 or the Pareto plot as in Figure B.6

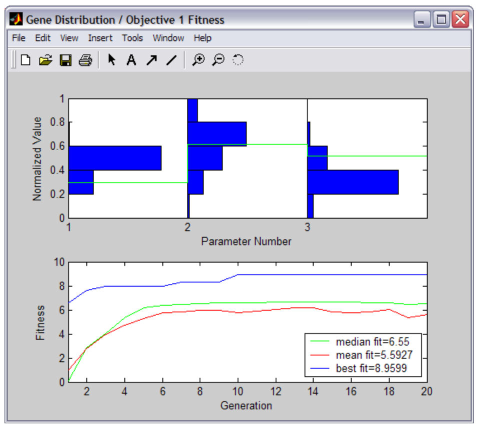


Figure B.5 Gene distribution plot

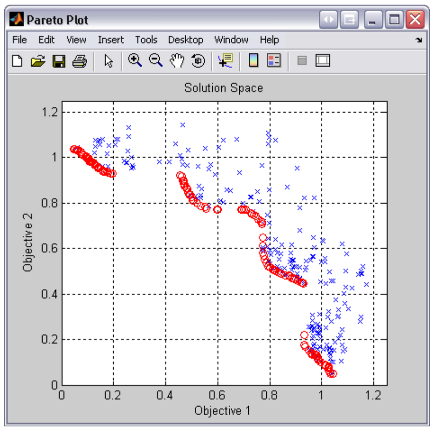


Figure B.6 2D Pareto plot

It is also possible to use a custom plotting routine on top of the distribution/fitness history plot and the Pareto plot by defining custom report plot handle `GAP.rp_crh`. The custom report plotting routine must have the following format without output return value.

```
f(P,GAP)
```

`P`	structure of current population
`GAP`	structure of genetic algorithm parameters

See also `distplot, paretoplot`

scale

Purpose Update scaling parameters and computes the scaled and aggregated fitness

Syntax fit = `scale(P,GAP)`

Arguments

`P`	structure of the input population
`GAP`	structure of genetic algorithm parameters

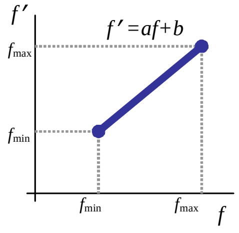
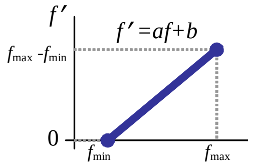
Value

`fit`	scaled and aggregated fitness

Description `scale` generate the scaled and aggregated fitness value (`P.fit`) based on the current `GAP` and the current population. Scaling operator is applied independently to each region in the multiple region case.

Given the current fitness values, each fitness values (`P.fit`) is penalized by multiplying the penalty function value (`P.pen`) generated from the diversity control routine. Then the maximum (f_{max}), minimum (f_{min}), average (f_{avg}), media (f_{med}) and standard deviation (f_{std}) of the penalized fitness value of the population in each region are found.

Depending on the value of scaling algorithm parameter `GAP.sc_alg`, different scaling method is used as in Table B.6.

Scaling algorithm number (`GAP.sc_alg`)	**Scaling method**	**Operation** f = original fitness f' = scaled fitness	
0	None	 f' $f'=af+b$ f_{max} f_{min} f_{min} f_{max} f	$a=1$ $b=0$
1	Offset scaling	 f' $f'=af+b$ $f_{max}-f_{min}$ 0 f_{min} f_{max}	$a=1$ $b=-f_{min}$

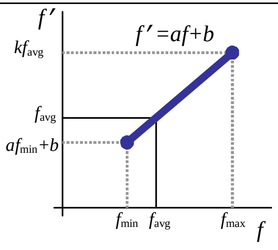
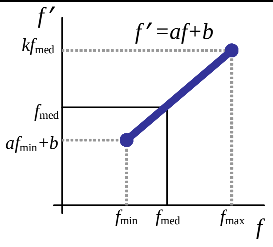
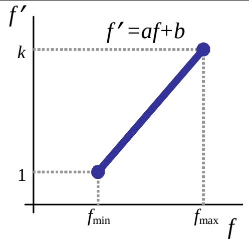
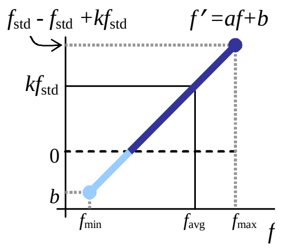
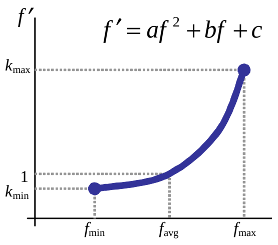
2	Standard linear scaling	 $a=\dfrac{(k-1)f_{avg}}{f_{max}-f_{avg}}$ $b=f_{avg}(1-a)$ $k=$ `GAP.sc_kln`
3	Modified linear scaling	 $a=\dfrac{(k-1)f_{med}}{f_{max}-f_{med}}$ $b=f_{med}(1-a)$ $k=$ `GAP.sc_kln`
4	Mapped linear scaling	 $a=\dfrac{k-1}{f_{max}-f_{min}}$ $b=-f_{min}\cdot a+1$ $k=$ `GAP.sc_kln`
5	Sigma truncation	 $a=1$ $b=-(f_{avg}-k\cdot f_{std})$ $k=$ `GAP.sc_cst`
6	Quadratic scaling	 $\begin{bmatrix}a\\b\\c\end{bmatrix}=\begin{bmatrix}f_{max}^2 & f_{max} & 1\\f_{avg}^2 & f_{avg} & 1\\f_{min}^2 & f_{min} & 1\end{bmatrix}^{-1}\begin{bmatrix}k_{max}\\1\\k_{min}\end{bmatrix}$ $k_{max}=$ `GAP.sc_kmxs` $k_{min}=$ `GAP.sc_kmns`

Table B.6 Scaling algorithms

If `GAP.sc_alg` = 0, scaling is not used.

If `GAP.sc_alg` = 1, offset scaling is used and f_{min} is mapped to 0 and f_{max} is mapped to $|f_{max} - f_{min}|$.

When `GAP.sc_alg` = 2, the fitness values are mapped so that the scaled fitness values also have same average fitness value as the original fitness value and the maximum fitness value is `GAP.sc_kln` times larger then f_{avg} .

The case of `GAP.sc_alg` = 3 is similar to the case of `GAP.sc_alg` = 2, except that median fitness value is used instead of average fitness value.

With `GAP.sc_alg` = 4, fitness values are linearly scaled such that f_{min} is mapped to 1 and f_{max} is mapped to `GAP.sc_kln`.

Sigma truncation is applied when `GAP.sc_alg` = 5. All the fitness values smaller than (f_{avg} < `GAP.sc_cst` f_{std}), where f_{avg} is the average fitness value and the f_{std} is the standard deviation of the fitness values, are mapped to negative values and therefore disregarded later by clipping to zeros. It is useful when there are few individuals with very small fitness value and most individuals have large fitness value.

If `GAP.sc_alg` = 6, quadratic scaling is used. This algorithm emphasizes the large fitness value and deemphasizes the small fitness value. The parameters of a quadratic function is found such that f_{max} is mapped to `GAP.sc_kmxs`, f_{avg} to 1 and f_{min} to `GAP.sc_kmns`, Then other fitness function values are mapped according to this quadratic function. `GAP.sc_kmns` is set to less than 1.

After applying the above scaling, all the negative fitness values are set to zeros, the fitness values are divided by the sum of all the fitness values. These final fitness values become the scaled fitness values that represent selection probabilities.

As the last step, the aggregate fitness values `P.fit` for chromosomes are obtained by summing all the objective functions using the objective function weight vector `GAP.owv`.

Examples

Given the fitness function vector `P.mfit` with three objective functions and five chromosomes as the following,

$$\text{P.mfit} = \begin{bmatrix} 4 & 5 & 10 & 10 & 11.1111 \\ 14 & -7.5 & 20 & 6.25 & 3.3333 \end{bmatrix}$$

If the penalty vector is

$$\text{P.pen} = [\ 0.5 \quad 0.8 \quad 0.6 \quad 0.8 \quad 0.9],$$

then the penalized fitness becomes

$$\begin{bmatrix} 2 & 4 & 6 & 8 & 10 \\ 7 & -6 & 12 & 5 & 3 \end{bmatrix}$$

If we apply standard linear scaling with the scaling factor (`GAP.sc_klin`) of 2, we have a = 1.5, b = -3 for the first objective and a = 0.5385, b = 1.9385 for the second objective to yield

$$\begin{bmatrix} 0 & 3 & 6 & 9 & 12 \\ 5.7077 & -1.2923 & 8.4000 & 4.6308 & 3.5538 \end{bmatrix}.$$

To make the fitness values non-negative, any negative fitness values are set to zero, that is,

$$\begin{bmatrix} 0 & 3 & 6 & 9 & 12 \\ 5.7077 & 0 & 8.4000 & 4.6308 & 3.5538 \end{bmatrix}.$$

Then the fitness values are normalized by dividing the fitness value by the sum of the fitness value of the corresponding objective.

$$\begin{bmatrix} 0 & 0.1 & 0.2 & 0.3 & 0.4 \\ 0.2560 & 0 & 0.3768 & 0.2077 & 0.1594 \end{bmatrix}$$

With the objective function weight [0.4 0.6], the aggregate fitness values are found to be

`P.fit` = [0.1536   0.0400   0.3061   0.2446   0.2557]

Reference D. E. Goldberg, *Genetic Algorithm in Search, Optimization, and Machine Learning*, Addison Wesley Publishing Company, January 1989

select

Purpose Select chromosomes from the population and form a mating pool

Syntax `[P_list,PL_size] = select(P,GAP)`

Arguments

`P`	structure of current population
`GAP`	structure of genetic algorithm parameters

Value

`P_list`	Parent list
`PL_size`	Parent list size

Description Selection operator picks chromosomes from the current population to construct a mating pool for reproduction. When multiple regions are used, selection is applied within each region. That is, if there are *m* chromosomes in a region, the selection operator picks the chromosome only from that region until *m* spaces of the mating pool are filled. There are two different selection methods in GOSET that one can choose from. They are roulette wheel selection and tournament selection.

In the selection operation, the aggregate fitness values (`P.fit`) are divided by the sum of the aggregate fitness value to yield the normalized aggregate fitness values.

Roulette wheel selection

Setting `GAP.sl_alg = 1` will activate roulette wheel selection. In the roulette wheel selection, the probability of an individual to be selected to the mating pool is proportional to the aggregate fitness (`P.fit`).

Tournament selection

Tournament selection is used if `GAP.sl_alg` is set to 2. In the tournament selection, `GAP.sl_nts` individuals are randomly chosen, and their aggregate fitness values (`P.fit`) are compared and the individual with best fitness value is selected to the mating pool. This procedure is repeated until the mating pool is occupied.

Illustrations of these selection operators are in Chapter 2.

Custom selection

Custom selection routine can be used instead of the two existing selection algorithms. This is specified by setting `GAP.sl_alg` to 3 and setting `GAP.dt_cah` with the handle of the custom function. The custom function must have the following format

`P_list = f(region,size,age,mfit,fit)`

`P_list`	indices of the individuals to become parents
`region`	the region number
`size`	number of individuals for the death list
`age`	vector describing ages of the individuals in population
`mfit`	array with raw fitness values of the individuals in the region
`fit`	vector of aggregate fitness values of individuals in the region

As an example of a custom algorithm, the Roulette wheel selection algorithm is written as an mfile called 'customselect.m' which is shown below.

```
% Custom select algorithm example – Roulette wheel selection algorithm
function plist = customselect(region,size,region_age,region_mfit,region_fit)

% determine the mating probability
Matprob = region_fit/sum(region_fit);

% create a mapping function for selection
map=cumsum(matprob);
% now do the selection
for i=1:size,
    choice=rand;
    j=1;
    while (choice > map(j))
        j=j+1;
    end
    plist(i)=j;
end
```

This mfile must be in the same folder as the fitness function file or in the GOSET Core folder. Then the custom file handle `GAP.sl_cah` is set to `@customselect`.

trimga

Purpose Randomly initialize the gene values and the regions of the individuals

Syntax `[x,f]=trimga(GAP,P,[D])`

Arguments

`GAP`	structure of genetic algorithm parameters
`P`	structure of a population
`D`	optional data required by fitness function

Value

`x`	revised solution
`f`	revised fitness function of the revised solution

Description The `trimga` operator uses the Nelder-Mead simplex algorithm to perform an deterministic optimization using the best individual from a GA as a starting point. The goal is to find a better solution in the vicinity of the obtained GA solution. The trimga only works with single-objective optimization problems. Gene range constraints are enforced by subtracting infinitity from the fitness function when the gene range goes outside of the prescribed limits. By default, for a single-objective optimization problem, trimga is executed automatically after the evolution process. Set `GAP.trimga` to 1 or 0 to enable or disable this feature.

unrndinit

Purpose Randomly initialize the gene values and the regions of the individuals

Syntax `P = unrndinit(P,GAP)`

Arguments

`P`	structure of current population
`GAP`	structure of genetic algorithm parameters

Value

`P`	structure of the updated population

Description `unrndinit` randomly generates chromosomes of the initial population.

First, the normalized gene values are randomly assigned as in the following,

$$P_{ng;j,k} = \texttt{rand}$$

where $P_{ng;j,k}$ represents the normalized gene value of *j*'th gene in the *k*'th individual and `rand` is MATLAB function that generates a random number between 0 and 1.

For the integer type gene, the normalized gene values are assigned with a discretized value between 0 and 1 such that they can represent correct integer values when mapped to actual gene values, that is,

$$P_{ng;j,k} = \frac{\texttt{fix}\ (\texttt{rand} \times \text{levels})}{\text{levels} - 1}$$

where $\text{levels} = P_{\max;j} - P_{\min;j} + 1$, and `fix` is a MATLAB function that rounds a number towards zero.

After this step, the actual gene values are determined according to their types by using `rawgene`.

If multi-regions are used, the chromosomes are distributed into regions by

$$P_{reg} = \texttt{ceil}\ (\texttt{rand} \times \texttt{GAP.nreg})$$

where `ceil` is a MATLAB function that rounds a number towards positive direction.

updateage

Purpose	Update the age of the each individual in the population	
Syntax	`newage = updateage(P)`	
Arguments	`P`	structure of current population
Value	`newage`	vector of new ages
Description	The age of each individual in the population is updated. The age of the individual survived from the previous generation increase by one, and the age of the new individual is set to one.	

updatestat

Purpose Update the statistic information of GAS

Syntax `GAS = updatestat(GAP,GAS,Pin)`

Arguments

`GAP`	structure of genetic algorithm parameters
`GAS`	structure of genetic algorithm statistics
`Pin`	structure of current population

Value

`GAS`	structure of updated genetic algorithm statistics

Description The current average fitness value, the median fitness value, the best fitness value, and the gene values of the best individual are added to `GAS.meanfit`, `GAS.medianfit`, `GAS.bestfit`, and `GAS.bestgenes` respectively. The number of total evaluation is updated to `GAS.ne`.

Appendix C. GOSET structures

P.[✶]	Description
P.fithandle	Handle to the fitness function
P.size	The number of individuals in the population
P.mfit	Unconditioned fitness function values (P.nobj × P.size)
P.fit	Fitness function values (1 × P.size)
P.eval	Fitness evaluation flag (1 × P.size) 0 : fitness is not evaluated 1 : fitness is evaluated
P.age	Age of each individual of the population in generations
P.ngenes	Number of genes in all chromosomes of an individual
P.min	GAP.gd_min
P.max	GAP.gd_max
P.type	GAP.gd_type
P.chrom_id	GAP.gd_cid
P.normgene	Normalized gene values (P.nobj × P.size)
P.gene	Gene values (P.nobj × P.size)
P.region	Geographic region (1 × P.size) of an individual
P.pen	Penalty function (1 × P.size) which is used for diversity control

GAP.[✶]	Description	Default
Fundamental parameters		
GAP.fp_ngen	No. of generations to evolve	100
GAP.fp_ipop	No. of chromosomes in initial population	100
GAP.fp_npop	No. of chromosome in the population	100
GAP.fp_nobj	No. of objective functions	1
GAP.fp_obj	Objective to optimize (0 for Multi-objective optimization)	1 / 0
Diversity control parameters		
GAP.dc_act	Diversity control usage flag 0: non-active 1: active	1
GAP.dc_alg	Diversity control algorithm	4
GAP.dc_spc	Diversity control space 1 : Parameter space 2 : Fitness function space	1
GAP.dc_mnt	Minimum threshold for algorithm 1	0.02
GAP.dc_mxt	Maximum threshold for algorithm 1	0.1
GAP.dc_ntr	No. of trials for algorithm 2	3
GAP.dc_mnb	Min no. of bins relative to pop. size for algorithm 2	0.5
GAP.dc_mxb	Max no. of bins relative to pop. size for algorithm 2	2
GAP.dc_dc	Diversity control distance const (Algorithm 3 & 4)	0.001
GAP.dc_nt	Diversity control test pop. size (Algorithm 4)	50
Scaling parameters		
GAP.sc_alg	Scaling algorithm 0 : none 1 : offset so minimum fitness is zero 2 : lin. scaling ($f_{avg} \rightarrow f_{avg}$, $f_{max} \rightarrow$ GAP.sc_klin×f_{avg}) 3 : lin. scaling ($f_{med} \rightarrow f_{med}$, $f_{max} \rightarrow$ GAP.sc_klin×f_{avg}) 4 : lin. scaling ($f_{min} \rightarrow 1$, $f_{max} \rightarrow$ GAP.sc_klin) 5 : sigma truncation 6 : quadratic scaling	1
GAP.sc_kln	Scaling factor for linear scaling algorithms	10
GAP.sc_cst	Scaling constant for sigma truncation	2
GAP.sc_kmxq	Max scaling factor for quadratic scaling ($f_{max}\rightarrow$GAP.sc_kmxq)	10
GAP.sc_kmnq	Min scaling factor for quadratic scaling ($f_{min}\rightarrow$GAP.sc_kmnq)	0.01
Selection algorithm parameters		
GAP.sl_alg	Selection algorithm 1: Roulette wheel 2: Tournament 3: Custom	2
GAP.sl_nts	No. of individuals used in a tournament	4
GAP.sl_cah	Function handle for the custom selection algorithm	[ ]
Death algorithm parameters		
GAP.dt_alg	Death algorithm 1: replace parents 2: random replacement 3: tournament on fitness 4: tournament on age 5: custom algorithm 6: random among 1 - 4	2
GAP.dt_nts	No. of individuals used in a tournament	4
GAP.dt_cah	Function handle for the custom death algorithm	[ ]
Mating and crossover parameters		

GAP.mc_pp	Percentage of pop. replaced by children	0.6
GAP.mc_fc	Fraction of chromosomes involved in crossover	1
GAP.mc_alg	Crossover algorithm 1 : Single point crossover 2 : Scalar simple blend crossover 3 : Vector simple blend crossover 4 : Scalar simulated binary crossover 5 : Vector simulated binary crossover 6 : Random algorithms	6/4
GAP.mc_gac	No. of gen. btw changing Algs for random crossover Alg	3
GAP.mc_ec	Tightness of distribution (η_c) for crossover algorithms 4 and 5	2
Mutation parameters		
GAP.mt_ptgm	Probability of a total gene mutation	▨
GAP.mt_ptgm0	Initial value of GAP.mt_ptgm	0.01
GAP.mt_ptgmf	Final value of GAP.mt_ptgm	0.001
GAP.mt_prgm	Probability of a relative partial gene mutation	▨
GAP.mt_prgm0	Initial value of GAP.mt_prgm	0.02
GAP.mt_prgmf	Final value of GAP.mt_prgm	0.002
GAP.mt_srgm	Standard deviation of relative partial gene perturbation	▨
GAP.mt_srgm0	Initial value of GAP.mt_srgm	0.3
GAP.mt_srgmf	Final value of GAP.mt_srgm	0.03
GAP.mt_pagm	Probability of a absolute partial gene mutation	▨
GAP.mt_pagm0	Initial value of GAP.mt_pagm	0.02
GAP.mt_pagmf	Final value of GAP.mt_pagm	0.002
GAP.mt_sagm	Standard deviation of absolute partial gene mutation	▨
GAP.mt_sagm0	Initial value of GAP.mt_sagm	0.1
GAP.mt_sagmf	Final value of GAP.mt_sagm	0.01
GAP.mt_prvm	Probability of relative vector mutation	▨
GAP.mt_prvm0	Initial value of GAP.mt_prvm	0.02
GAP.mt_prvmf	Final value of GAP.mt_prvm	0.002
GAP.mt_srvm	Sandard deviation of relative vector mutation	▨
GAP.mt_srvm0	Initial value of GAP.mt_srvm	0.3
GAP.mt_srvmf	Final value of GAP.mt_srvm	0.03
GAP.mt_pavm	Probability of absolute vector mutation	▨
GAP.mt_pavm0	Initial value of GAP.mt_pavm	0.02
GAP.mt_pavmf	Final value of GAP.mt_pavm	0.002
GAP.mt_savm	Standard deviation of absolute vector mutation	▨
GAP.mt_savm0	Initial value of GAP.mt_savm	0.1
GAP.mt_savmf	Final value of GAP.mt_savm	0.01
GAP.mt_pigm	Probability of integer gene mutation	▨
GAP.mt_pigm0	Initial value of GAP.mt_pigm	0.2
GAP.mt_pigmf	Final value of GAP.mt_pigm	0.01
Gene repair parameters		
GAP.gr_alg	Gene repair algorithm 1 : Hard limiting 2 : Ring mapping	1
Migration parameters		
GAP.mg_nreg	No. of geographic regions the population is distributed	GAP.fp_npop/100
GAP.mg_tmig	Time between migrations in generations	GAP.fp_ngen/20
GAP.mg_pmig	Probability of an individual to migrate	0.1
Evaluation parameters		
GAP.ev_bev	Block evaluation 0 : evaluate an individual 1 : evaluate all the individual	0
GAP.ev_are	Fitness reevaluation option 0: evaluate only the unevaluated 1: evaluate all	0
GAP.ev_ssd	Supplementary data 0: Pass P.gene 1: pass P.age, P.mfit, P.region	0
GAP.ev_pp	Use parallel processing true=yes false=no	
GAP.ev_npg	Number of evaluation groups for parallel processing	
Elitism Parameters		
GAP.el_act	Elitism activation flag	1
GAP.el_fgs	Fraction of generation to pass before starting random search	0
GAP.el_fpe	Fraction of pop. protected as elite for multi-objective optimization	▨
GAP.el_fpe0	Initial value of GAP.el_fpe	0.2
GAP.el_fpef	Final value of GAP.el_fpe	0.8

Random search parameters		
GAP.rs_fgs	Fraction of generation to pass before starting random search	0.5
GAP.rs_fps	Fraction of total population size used in random search	0.1

GAP.rs_srp	Standard deviation used in relative perturbation	0.3
GAP.rs_sap	Standard deviation used in absolute perturbation	0.1
GAP.rs_frp	Fraction of time that relative random perturbations are used. Absolute random perturbation is used for the rest of the time.	0.7
GAP.rs_fea	Fraction of generations on which to execute the algorithm	0.2
Reporting parameters		
GAP.rp_lvl	Reporting level -1: no report 0: text only 1: plot & text	1
GAP.rp_gbr	Generation between reports	5
GAP.rp_crh	Function handle for custom reporting algorithm	[ ]
GAP.so_ob	Objective function number used for output sorting	1
Objective plot parameters		
GAP.op_list	List of objectives to make objective plots for	[1]
GAP.op_style	Style for each objective 0: logarithmic 1: linear	[1 … 1]
GAP.op_sign	Sign of fitness for each objective -1: neg 1: pos/mixed	[1 … 1]
GAP.op_srt	Objective number that the output plot should be sorted by	1
GAP.dp_type	Distribution plot type 1: plot individuals 2: plot histograms	1
GAP.dp_np	Maximum number of individuals to plot	100
GAP.dp_res	Number of bins in distribution plot for type 2	20
Pareto plot parameters		
GAP.pp_list	List of 2 or 3 objectives to be used in Pareto plot	[ ]/[1, 2]
GAP.pp_xl	x-axis label	'Objective 1'
GAP.pp_yl	y-axis label	'Objective 2'
GAP.pp_zl	z-axis label	'Objective 3'
GAP.pp_title	Pareto plot title	'Solution space'
GAP.pp_style	Style for each objective 0: logarithmic 1: linear	[1 … 1]
GAP.pp_sign	Sign of fitness for each objective -1: neg 1: pos/mixed	[1 … 1]
GAP.pp_axis	Axis limits for Pareto Plot	[ ]
Gene definition parameters		
GAP.gd_min	Row vector of minimum gene values	Defined by the user
GAP.gd_max	Row vector of maximum gene values	
GAP.gd_type	Row vector of gene types 1: integer 2: linear 3: logarithmic	
GAP.gd_cid	Row vector of chromosome ID number	
Trim GA parameter		
GAP.trimga	Execute TRIMGA after the evolution (for single-objective optimization only)	1

GAS.[✶]	Description
GAS.cg	Current generation number
GAS.medianfit	The median fitness values of each objective
GAS.meanfit	The average fitness values of each objective
GAS.bestfit	The best fitness values of each objective
GAS.bestgenes	The best gene values for each objective over the generations
GAS.ne	The number of the total objective function evaluations

Abbreviation list No.: Number Min: Minimum Max: Maximum Pop.: population Gen.: Generation Alg: Algorithm Neg: Negative Pos: Positive