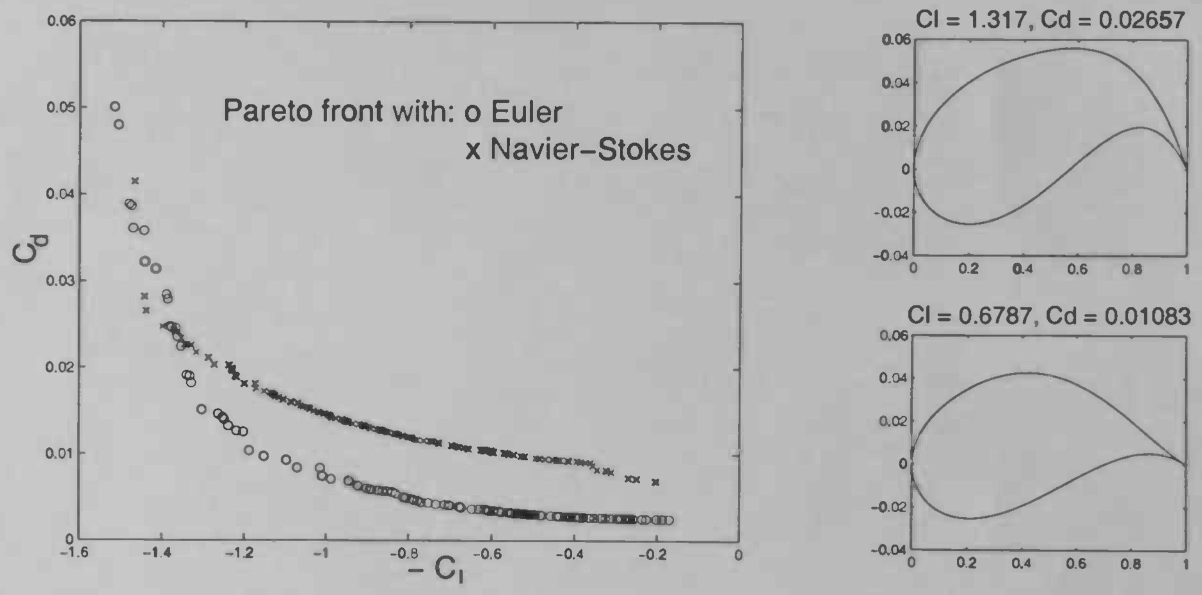
Genetic algorithms in multiobjective design optimisation

Simone De Kleermaeker



Department of
Mathematics

RuG

Master's thesis

Genetic algorithms in multiobjective design optimisation

Simone De Kleermaeker

University of Groningen
Department of Mathematics
P.O. Box 800
9700 AV Groningen

October 2000

Contents

Abstract

Optimisation is a challenge for computerized multidisciplinary design. With multidisciplinary design optmisation there are several objectives to optimise, originating from different disciplines. Therefore no obvious method to splice the multiple objectives into a single objective exists. Multiobjective optimisation is a method that can deal with multiple objectives.

The genetic algorithm is a global optimisation method, which is inspired by natural evolution. It uses a population of solutions that evolves to better solutions with help of recombination and mutation. A genetic algorithm can cope with a design space that is, for example, discontinous or convex. That and the global character of the genetic algorithm make it a suitable algorithm to solve multiobjective optimisation problems.

In this report an introduction to optimisation problems and genetic algorithms is given. Furthermore, the results are shown of two multiobjective optimisation problems: parameter choice in time integration methods and airfoil design.

Chapter 1

Introduction

The global subject of this report is multidisciplinary design optimisation. Such optimisation deals with several objectives, which originate from different disciplines. The objectives represent the properties of the design to be optimised. For example, when designing a wing, the aerodynamic lift should be maximised, the structural mass should be minimised and the volume of the wing should be able to contain a certain minimal amount of fuel. These objectives originate from different disciplines. This example illustrates the need for multiobjective optimisation when dealing with multidisciplinary design optimisation. Design optimisation in general can be performed in different ways. A single objective (SO) can be formulated, which is to be optimised in a single objective optimisation (SOO). Also several, or multiple, objectives can be formulated. There are different ways to deal with multiple objectives. The different objectives can be spliced into one single objective, for instance by a weighted sum of the objectives (multiple objective optimisation). This however has the disadvantage that the relative importance of the different objectives is fixed a priori by the values of the weighting factors. With multidisciplinary design it is preferable to optimise the different objectives simultaneously in the optimisation, and to decide about the importance of each of the objectives after the optimisation. This is achieved by multiobjective optimisation (MOO), which will be the main subject of the report.

This study tries to find an answer to the question: Can a genetic algorithm be used to solve a MOO problem and how well does such an algorithm perform on a MOO problem.

Multidisciplinary design optimisation (MDO) leads to multiobjective optimisation (MOO) if the objectives can not obviously be gathered in a single objective with a multiple objective optimisation. In the aerodynamic multiobjective design optimisation the objectives may be discontinuous, therefore the distribution of the objectives can be quite unexpected and the resulting aerodynamic optimisation problem can be very difficult. In this situation, a global search algorithm is indispensable. More detailed information on design optimisation is presented in chapter 2.

In a literature survey on Multiobjective Design Optimisation [13] some design optimisation methods have been examined. With the purpose of this research project in mind, the optimisation methods are judged. It concentrates on multiobjective aerodynamic design optimisation and especially airfoil shape optimisation. Some methods are introduced in chapter 2. For more information on these methods the reader is referred to [13].

Beforehand the expectation existed that the genetic algorithm (GA) would be one of the optimisation algorithms fit for aerodynamic MOO. GAs are a specific type of evolutionary algorithms (EAs). The idea behind EAs is based on natural evolution. The GA handles a population of solutions, i.e. instead of one point in the search space, a whole population of points, or *individuals*, is examined. The GA is a global search algorithm. All the possible solutions have to be *coded* into *genes*. *Chromosomes*, which are composed of these genes, represent the individuals of the population. This population has to *evolve* to several optima in the search space in the process of the GA. To make the population evolve, recombination operators are created. With these operators *parents* can be selected from the population who can *mate* to produce *children* who can replace some of the old individuals in the new population. The objectives determine the *fitness* of an individual which influences the probability of an individual to be selected as parent. A GA should be *tuned* in its coding and genetic operators for each problem it is applied to. More detailed information on the genetic algorithm can be found in chapter 3.

Different implementations of optimisation algorithms in general and genetic algorithms in particular are discussed in chapter 4. The optimisation algorithms of Matlab are looked at and implementations of several GAs are discussed.

Any algorithm should be validated to learn what can be expected from this algorithm. The test functions should be chosen carefully. They should test all aspects of the algorithm and all possible difficulties that may rise from the type of optimisation problem the algorithm will have to optimise. In Chapter 5 a genetic algorithm is validated.

When the algorithm works properly, some testing is required to find the right settings for the parameters of the algorithm. With a genetic algorithm such parameters could be, for example, the size of the population and the number of generations. In chapter 6 some test functions are presented and discussed.

Once validated and tuned, the genetic algorithm can be used for real optimisation problems. A constrained multiobjective optimisation problem searching for the optimal parameter values for a Runge Kutta time discretisation scheme is presented in chapter 7.

In chapter 8 multiobjective design optimisation of the aerodynamic properties of an airfoil is presented. CFD calculations are used for the evaluation of these aerodynamic properties. This optimisation makes use of the parallel properties of a genetic algorithm. In this chapter also some suggestions are made on how to continue with multidisciplinary design optimisation problems.

Finally, conclusions and recommendations are given in chapter 9.

Chapter 2

Design Optimisation

When designing a product, numerical simulations can help with the search for the best design in the space of all possible designs. During this process, not only the requirements should be met, but also some of the properties of the product should be optimised. Optimisation algorithms can be used to do the latter job. The properties that are to be optimised should be formulated in objectives and the requirements should be formulated as constraints, which most optimisation algorithms can handle. The fitness of a solution can be calculated using the objectives. Each objective supplies an separate fitness value for a solution. The task is to find an optimisation algorithm which is the best to use in a specific design optimisation problem. There are many different optimisation algorithms. Many articles deal with comparing different algorithms (see [13] and [17]). When searching for the ideal optimisation algorithm, the "No free lunch" theorem will come up: There simply is no such thing as the ideal optimisation algorithm for all optimisation problems. With each new problem it should be checked which algorithm is best to use. There is however an algorithm that in many cases will come to the desired result. This is the genetic algorithm (GA). A GA can be very inefficient when used in an inappropriate optimisation problem, but in time it will often lead to the right answers. Advantage of the GA is that no knowledge is needed of the design space in which the optimal solution is searched for. The design space is the feasible region, i.e. with solutions which satisfy the constraints, within the parameter space. The design space can, for example, be convex or discontinuous, the functions that are to be optimised do not have to be smooth. A GA can cope with all these things. The GA is further discussed in chapter 3.

So in design optimisation problems one or more objectives have to be optimised. This can be achieved in different ways. If there is only one objective to optimise, the optimisation is called single objective optimisation (SOO). A problem with more than one objective to optimise can be treated as a multiple objective optimisation problem or as a multiobjective optimisation (MOO). A multiple objective optimisation problem is a problem in which all the objectives are combined into a single objective, for example by a weighted sum. With MOO the different objectives are all optimised simultaneously.

2.1 Single and multiple objective optimisation

In SOO the solution of the optimisation is the best value (maximum or minimum) of the assumed well-defined objective, i.e. a cost function or utility. There are different optimisation strategies to find such a optimum. Some examples are: parametric optimisation, unconstrained

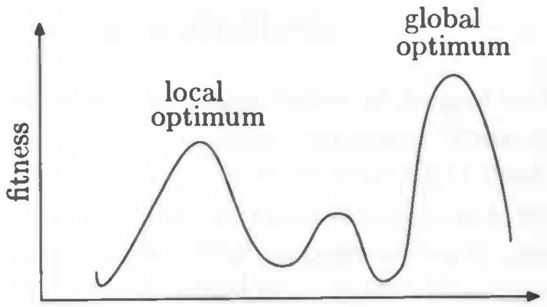


Figure 2.1: A gradient based method will not find the global optimum from many initial points.

or constrained optimisation and least squares optimisation. Often SOO algorithms are gradient-based methods (GM). This works for optimisation problems with only one local and therefore one global optimum. But if there are local optima which are different from the global optimum a gradient-based optimisation method could get stuck in a local optimum and thus will not find the global optimum. An example of such an objective is given in figure 2.1.

With multiple objective optimisation, the solution of the optimisation problem still lies in the optimum of one objective. Examples of multiple objective optimisation are weighed sum methods and the multiple β concept [16]. This last method is a concept for formulating a MOO as a SOO and hence makes it into a multiple objective optimisation.

2.2 Multiobjective optimisation

The simultaneous optimisation of multiple, possibly competing, objectives deviates from single objective optimisation because there is no single, perfect solution. Instead, multiobjective optimisation problems tend to be characterised by a family of alternative solutions which must be considered equivalent in the absence of information concerning the relevance of each objective relative to the others. Multiple solutions arise even in the simplest non-trivial case of two competing objectives, where both are unimodal and convex functions of the decision variables [6].

Thus for MOO a different definition of optimality is needed, one that respects the integrity of all the separate objectives. The concept of Pareto optimality helps defining such optimal solutions in a rational way.

Definition 2.2.1 ***Pareto Optimality*** *A solution is called non-dominated if there is no solution which is better on at least one objective and no worse on all the other objectives.*

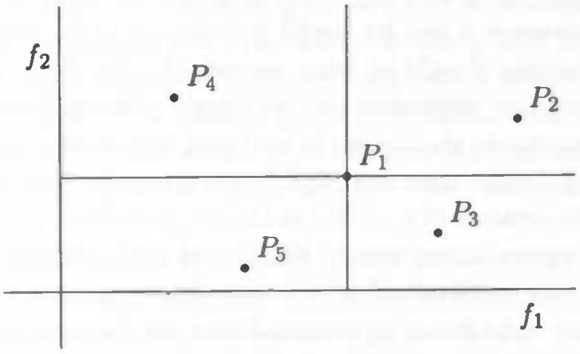


Figure 2.2: Pareto optimality.

So an individual is non-dominated if all the other solutions have at least one objective for which the solution is worse. In figure 2.2 the fitness values of some solutions of a minimisation problem have been plotted. Solution P_1 dominates the solution P_2. P_1 is dominated by solution P_3 and is equivalent to the solutions P_3 and P_4.

All Pareto optimal points found by the optimisation plotted in the objectives space together form the Pareto front. The *real* Pareto front is the front of analytical non-dominated points of the optimisation problem. The Pareto front is the front as it is reached with the optimisation, and is formed by just the non-dominated points found with that optimisation. From this front a single non-dominated solution can eventually be chosen as *the* optimal solution, representing the best design. By searching for the entire Pareto front in the optimisation instead for the one optimal solution that will in the end become the design used, the moment of decision making is postponed until after the optimisation. This is useful because of several reasons, e.g. before the optimisation it is not known what the shape of the Pareto front will be and were it will be positioned in the objectives space. After the optimisation this information is known and a final decision can be made.

Ranking based on the definition of Pareto optimality of a set of solutions works as follows: First assign all the non-dominated solutions rank 1, then remove them from the set. The non-dominated solutions from this smaller set are assigned rank 2. Continue this procedure until there are no more individuals left in the set. In figure 2.2 solution P_5 is assigned rank 1, solutions P_1, P_3 and P_4 are assigned rank 2 and solution P_2 has rank 3.

For most optimisation algorithms it is necessary to keep track of the dominated solutions, the solutions with rank > 1, to find better solutions (see chapter 3). Therefor a rank is assigned to them as well. The Pareto front is visualised by a plot in the objective space of the objectives of all the individuals of the set of all solutions with rank 1. From this Pareto front, it is up to the designer to choose the best design. This could be done with the use of yet another objective or based on experiences of the designer.

2.3 Some optimisation algorithms

Once a good method of comparing the fitness values of designs with each other is established, the search for a suitable optimisation algorithm can begin. Some optimisation algorithms will be mentioned here, with a short explanation of the working of these algorithms.

Random search is a simple method. It explores the search space by randomly selecting solutions and evaluating their fitness. This strategy is rarely used by itself in optimisation problems. In testing however it can be useful to compare other algorithms with random search. Any optimisation algorithm should at least perform better than random search, otherwise the trouble of implementing this algorithm can be saved by using random search.

In gradient-based methods the design is updated iteratively in the direction of the steepest ascent from the initial design (the hill climbing strategy). This only works well if the initial point lies in the neighbourhood of a global optimum, provided that the gradient is well defined. Therefore, the design optimisation should be started with various initial points to determine if a consistent optimum can be obtained with reasonable assurance that this is a global optimum.

Simulated annealing (SA) is an algorithm based on the analogy between the simulation of the annealing of solids and the problem of solving large combinatorial optimisation problems. Basically, in this method, a design is replaced if a randomly disturbed version of this design is better. So far this strategy is still hill climbing and could end up with local extrema as easily as GM does. But SA also accepts a poorer design with certain probability related to the Boltzman distribution under the specified annealing schedule. This probability declines with time. In the analogy the probability to stay in a particular solution depends directly on the energy of the system and on its temperature. This probability is formally given by Gibbs' law: $p = e^{-\frac{E}{kT}}$, where E stands for the energy, k is the Boltzman constant and T is the temperature. With SA, the energy is given by the difference of the fitness function between the new and old solution and the decrease of the temperature is related to the increasing of the number of generations passed. So the new solution is retained with probability: $p = e^{-\frac{f(y)-f(x)}{kt}}$, where t stands for the time passed. Therefore, there is a corresponding chance for the design to get out of a local extreme in favour of finding a better, more global one.

These optimisation techniques are difficult to extend to the *true* multiobjective case, because they were not designed with multiple solutions in mind. In practice, multiobjective problems have to be re-formulated as single objective problems prior to optimisation leading to the production of a single solution per run of the optimiser [6].

Evolutionary algorithms are well-suited to multiobjective optimisation. Like simulated annealing algorithms they are based on nature. Multiple individuals can search for multiple solutions in parallel. The ability to handle complex problems, involving features such as discontinuities, multi-modality, disjunct feasible spaces and noisy function evaluations, reinforces the potential effectiveness of EAs in multiobjective search and optimisation. This is perhaps a problem area where Evolutionary Computation really distinguishes itself from its competitors [6].

The genetic algorithm is an evolutionary algorithm. The algorithm starts with a random population of designs, the individuals. With reproduction mechanisms this population evolves generation after generation to a final population that contains individuals with improved fitness values. This algorithm is further described in chapter 3.

Hybrid algorithms are combinations mostly of a global search algorithm and local optimisation algorithms. This works well, especially if some knowledge on the search space is present.

In that case the hybrid algorithm exploits the global perspective of, for example, a GA and the convergence of the problem-specific technique, for example the hill climbing technique.

2.4 Design optimisation

Design optimisation problems often deal with multidisciplinary objectives. For example, when designing a wing the lift produced by this wing as well as the strength of the wing should be considered. These are two objectives from two different disciplines for which it is not obvious how to splice them into a single objective with weights to get a multiple objective optimisation problem. Hence an algorithm for MOO should be used. This leads to multidisciplinary multiobjective optimisation (MDMOO). Because of this multiobjective character of MDMOO, this type of problems need optimisation algorithms that are developed with multiple solutions in mind, as is discussed in section 2.2.

In design optimisation involving CFD analysis, the objectives depend on CFD results. Consequently, the objectives are computationally expensive to evaluate. In the case of GAs, the computational costs are proportional to the number of individuals and the number of generations.

When dealing with a design optimisation problem, constraints may provide restrictions to the search space. There are many ways to handle constraints. In chapter 7 an example is given of an optimisation problem with a single constraint, the Runge-Kutta problem.

2.5 The choice

Three properties of the GA make the GA a suitable algorithm for multidisciplinary design optimisation. These properties are its ability of finding a Pareto front, the way it can handle MDMOO with constraints and its flexibility in the type of search space in which it can operate. Papers on this subject (see [13]) suggest the same choice and show that a GA is well capable of handling MOO.

An optimisation run with a GA needs *population size* × *number of generations* objective evaluations. This number of evaluations for a specific optimisation problem to get near the Pareto front could get large. When, for example, CFD calculations are necessary for the calculation of the fitness of the individuals optimisation runs can get computationally expensive and time consuming. However, the parallelisation possibilities of GAs may bring some improvement in this respect.

Chapter 3

Genetic Algorithms

3.1 Key aspects of genetic algorithms

GAs are directly derived from the working of natural evolution. First some terminology derived from biology used in evolutionary computation is introduced. A population consists of a number of individuals. A chromosome is the set of all the genes of a specific individual. Chromosomes are often used as a synonym for individuals. The genes code the characteristics of an individual. A gene is made up of several alleles and these alleles are the design parameters of the design represented by the individual. The gene pool is the set of all alleles present in a population. The diversity of the individuals in the population is directly determined by the size of the gene pool.

$$\text{Chromosome A:}\quad \overbrace{a_1\ a_2}^{\text{gene 1}}\ \underbrace{a_3\ a_4\ a_5}_{\text{gene 2}}\ \dots\ \overbrace{a_{m-1}\ a_m}^{\text{gene } p},\quad a_i \in (\text{gene pool})_i$$

Figure 3.1: A chromosome is an array of m alleles.

The GA deals with one generation of a population at a time. Designing a chromosome in such a way that all possible designs can be represented by its alleles, is the first, but certainly not the most straightforward part of developing a GA. Secondly the objectives expressing the desired properties of the design must be defined. Finally the set of reproduction operators that work on the alleles and genes of the individual must be determined.

With the objectives the fitness value per objective of an individual, i.e. a chromosome consisting of alleles, can be calculated. With these fitness values the rank of an individual with respect to the current population can be determined.

$$Chromosome \overset{\text{objectives}}{\Longrightarrow} \begin{bmatrix} fitness_1 \\ \dots \\ fitness_n \end{bmatrix} \overset{\substack{\text{Pareto} \\ \text{optimality}}}{\Longrightarrow} rank$$

In a GA a random initial population is generated and the fitness values and rank of the individuals are calculated. The evolution consists of a fixed number of generations. The first step of each generation consists in selecting individuals to be parents in the reproduction procedures. A selection procedure can have criteria based on, for example, the fitness values or the rank of an individual. For reproduction different types of recombination and mutation operators can be used. After reproduction the fitness values of the new individuals, the children, are evaluated. Depending on the fitness values of parents and children it is decided which parents and children will go into the next generation.

The selection mechanism compares the individuals in the population with each other. For SOO and multiple objective optimisation comparing individuals on their fitness is a straightforward job. With MOO this comparison is done with the help of ranking based on some kind of definition of optimality, e.g. the frequently used Pareto optimality mentioned in section 2.2. The selection mechanism selects parents from the population and determines which individuals should be recombined with each other. There are many different selection methods to use. The roulette wheel selection is a commonly used selection method. With this method each individual is selected as a parent with a chance related to the fitness or rank of the individual (see [13]). Other selection methods will be mentioned later in this chapter.

Recombination is a key operator for natural evolution. The most common form of recombination is one-point crossover. In a crossover, two chromosomes are each cut at one point and the parts are spliced.

$$\begin{array}{llcll} \text{parent A} & a_1\ a_2\ a_3\ a_4\ a_5 & & \text{child 1} & a_1\ a_2\ b_3\ b_4\ b_5 \\ & \quad\updownarrow \text{ crossover point} & \overset{\text{crossover}}{\Longrightarrow} & & \\ \text{parent B} & b_1\ b_2\ b_3\ b_4\ b_5 & & \text{child 2} & b_1\ b_2\ a_3\ a_4\ a_5 \end{array}$$

Figure 3.2: Crossover.

There are many variations on this one-point crossover, some of which are mentioned later on in this chapter. The effect of recombination is very important because it allows characteristics from two different parents to be combined.

Mutation is an other way to get new chromosomes. It is composed of randomly changing the value of one or more alleles. It is not a very common operator in natural evolution, but still a very powerful operator. Mutation has two important roles in simple GAs. One is to provide the capability to effectively refine sub-optimal solutions. Another is to maintain population diversity by re-introducing in the population the alleles lost by the repeated application of crossover. With mutation the population diversity can be very well preserved. Because of the random introduction of new alleles the refining capabilities of a mutation operator may not be very good for every class of problems. In that case a mutation operator could be more effective when the frequency of the mutation decreases with the increasing number of evolved generations. This way, individuals which have evolved towards good solutions do not loose their favourable properties through mutation.

3.2 Basic ingredients of GAs for multiobjective optimisation

The main aspects of a simple GA are the coding of the individuals, a selection method, a crossover and mutation operator and a ranking procedure. All these aspects should be more or less adapted to the specific optimisation problem. Some examples of different methods and operators are listed in the next subsection.

3.2.1 Tuning the simple genetic algorithm

1. Coding: The user should select the smallest alphabet for coding that permits a natural expression of the problem [10]. The most common types of coding are binary coding of the alleles and floating point coding of the alleles. But any type of coding with a specific alphabet can be used if this will work better with the optimisation problem.

   Binary coding is more or less derived from natures alleles, which can have only four different values (G,T,C and A). Binary chromosomes are generally longer than floating point chromosomes, because genes, which are groups of several alleles, are used to represent one parameter (see figure 3.4). These genes are translated into floating points using decode functions. With the floating point coding used in this report one allele represents one parameter. Because binary chromosomes are long, building blocks are more likely to arise with this coding. Building blocks are alleles close to each other, likely to still lie close together after crossover (see figure 3.3). These groups of alleles can represent certain favourable properties of the design. The building block hypothesis states that the existence of these blocks in a chromosome does indeed lead to better performance [10]. The bitwise recombination operators are designed for binary chromosomes and not for floating point chromosomes. By assigning different number of alleles for different parameters, the chance on crossover and mutation can be influenced per variable, because longer genes have bigger chance to hold the allele that is selected for mutation or crossover point then smaller genes.

$$\begin{array}{lcll}
\text{parent A } 110001001110111011 & & \text{child 1} & 110011110\underbrace{011010}_{\text{Building block}}111 \\
\updownarrow \text{ crossover} & \overset{\text{crossover}}{\Longrightarrow} & & \\
\text{parent B } 001011110\underbrace{011010}_{\text{Building block}}111 & & \text{child 2} & 001001001110111011
\end{array}$$

Figure 3.3: Building blocks.

   With floating point parameters the floating point representation is intuitively closer to the problem space. Therefore it is easier to design meaningful recombination operators and other operators incorporating problem specific constraints. Simple analyses seem to suggest that enhanced scheme processing is obtained by using alphabets of low cardinality. Empirical findings however show that real, or floating point, coding has worked well in a number of practical problems. This means that the type of coding appropriate for an optimisation depends on the problem at hand.

$$\begin{array}{ll}
\text{floating point coding:} & \underbrace{0.8\ 0.4}_{\text{property 1}}\ \overbrace{0.6\ 0.1\ 0.7}^{\text{property 2}}\ \underbrace{0.3\ 0.5}_{\text{property 3}} \\
\text{binary coding (of property 2):} & \underbrace{110}_{\text{gene 1}}\ \overbrace{001}^{\text{gene 2}}\ \underbrace{111}_{\text{gene 3}} \quad \xrightarrow{\textit{decoding}} \quad 0.6\ 0.1\ 0.7
\end{array}$$

Figure 3.4: Floating point and binary coding.

2. Selection: The roulette wheel selection was already mentioned in section 3.1. Besides

the fitness values and the rank of individuals, selection decisions can also be made on the basis of constraint violations [18]. The two-branch tournament does not require the non-dominated ranking approach nor additional fitness manipulations. It is organised so that designs compete on one or two objectives. As the name implies the selection procedure is divided into two branches, both branches select half of the parents needed for recombination. In one branch of the tournament the selection is based on the first objective; the other branch evaluates individuals on the second objective. Each branch can select at most one copy of an individual as a parent ([3],[13]). The binary tournament selection is another selection method which will be used in chapter 4 and explained in detail in appendix B.

branch 1:	select strings x_1 and x_2 →	$f_1(x_1) < f_1(x_2)$?	→ x_1 survives, x_2 discarded
		else	→ x_2 survives, x_1 discarded
branch 2:	select strings x_1 and x_2 →	$f_2(x_1) < f_2(x_2)$?	→ x_1 survives, x_2 discarded
		else	→ x_2 survives, x_1 discarded

Repeat until enough parents are selected.

Figure 3.5: Scheme of two-branch selection.

3. Crossover: As mentioned before, one-point crossover uses one crossover site (see figure 3.2). A multi-point crossover is a variation of this operator and uses multiple crossover sites. Another alternative to the standard one point crossover is the extended intermediate crossover (EIR)[20].

4. Mutation: This is an operator that consists in randomly changing part of the chromosome. There are many ways to do this. When using binary coding, mutation can be performed on a bit or word level, i.e., directly on the bits of the binary coding or on the integer number in the interval $[0, 2^n - 1]$, n being the number of bits adopted for each variable. In the latter case, the mutation operates on whole genes instead of alleles.

5. Ranking: With non-dominated sorting [7] the individuals are assigned a rank equal to the number of individuals which dominate it. This rank is easy to adjust when individuals enter or leave the population. Pareto ranking is used most often. With this method non-dominated solutions within the feasible region in the objectives space give the Pareto-optimal set. This ranking is described in more detail in section 2.2.

3.2.2 Improvements to a simple genetic algorithm

A simple GA needs a lot of generations (order 100) or a large population (order 100) to get in the neighbourhood of the *real* Pareto front. Therefore, the efficiency of the GA should be improved (item 1). A GA is efficient if the there are not many generations necessary for a population to converge towards the *real* Pareto front. The premature convergence of the population to part of the Pareto front or local optima is called genetic drift. This causes a loss of diversity of the gene pool and is a bad influence to the distribution of the points on the found Pareto front, so it should be avoided (item 2). The individuals of a population which are non-dominated could be used to spread the good properties they hold through the elitism mechanism (item 3).The GA could be combined with other optimisation methods to get the best of both methods (item

4). A good way to deal with constraints should be found (item 5). And the parallel properties of the GA should be exploited (item 6).

Some points of attention on improvements to a simple genetic algorithm are listed below.

1. Efficiency: The two-branch tournament GA mentioned in subsection 3.2.1 makes efficient use of the GAs population-based search. The computational effort of generating an approximation to the Pareto front becomes comparable to that of traditional approaches. Using the selection operator instead of the object function to perform multiobjective design is the significant difference of this approach. ARange GA [1] is a new genetic search algorithm which adapts the searching range according to the optimisation situation and makes it possible to obtain highly accurate results effectively.

2. Actions against genetic drift: Fitness sharing [7] lets individuals that are similar to one another mutually decrease each other's fitness by competing for the same resources. In tournament slot sharing [15] the genotypic distance between the individuals influences its probability to enter a tournament in the same way as with the roulette wheel selection. A replacement rule for reproduction procedures can be set: parents are replaced only when the rank of one of their offspring is no worse than the best rank of the parents. In the algorithm TSPspea mentioned in chapter 4 this fitness sharing is implemented in the form of a clustering function (see appendix B). In the distributed GA (DGA) [5] a subdivision of the population into semi-isolated subpopulations (*demes*) is used. This helps to maintain diversity of the gene pool. To avoid the premature convergence of a population on an island, an agent can be added to supervise the operation of the parallel GA.

3. Elitism mechanism: Another method to improve the convergence rate of the algorithms is the elitism mechanism. This mechanism can consist of keeping a separate subpopulation with copies of the best individuals from the population. This elite group can be placed in the next generation, randomly replacing children. Another form of this mechanism is to copy all the individuals from the old generation which would be non-dominated in the new generation to the new one. Or an assigned percentage of parents can be selected from this elite group to take part of the reproduction procedures. In chapter 4 the non-dominated set in the algorithm TSPspea functions as a group of elite-individuals.

4. Hybrid schemes: The most favourable capabilities of GAs might be favourably exploited through a proper hybrid scheme, combining them with an optimisation technique of different nature, so that the good features of both methods can be utilised. This optimisation technique should be more efficient in locally improving the fitness values then a GA. For example, a gradient-based optimisation routine, such as the hill climbing strategy, could function as such a refinement of the GA.

5. Constraint handling: One way to think of constraints is as high-priority or hard objectives, which must be jointly satisfied before the optimisation of the remaining, soft objectives take place. Each solution that does not satisfy a constraint is rejected. An example of such constraint handling is given in chapter 7. Another way is to combine the constraints with the objectives through a penalty function. The penalty functions give a positive contribution to the fitness value when the constraints are satisfied and a negative contribution when the constraints are violated. Similar penalty functions can be combined with the rank of an individual.

6. Parallelisation: The GA is a highly parallelisable process. Nearly all operations on the individuals of one generation are implicitly parallel. A parallel implementation based on the master-slave principle can be used: the master calculates the genetic operations and the slaves compute the fitness values. The amount of communication necessary is rather small. An example of such a parallel implementation is presented in chapter 8. The GA is well suited for implementation on a network of workstations.

   Some algorithms which exploit this possibility to maintain diversity of genes in the population, but were not used in this report, are the distributed GA, the island model and the stepping stones model. With the distributed GA (DGA), mentioned earlier with item 2, the only communication required between processors with this algorithm, involves migration from one neighbouring *deme* to another, at intervals of several generations. Similar are the island model and the stepping stones model [11]. In the island model all the *demes* are connected with each other, in the stepping stones model every *deme* is only connected with two neighbours (the *demes* lie in a circle). Migration can only take place between *demes* that are connected with each other.

In chapter 4 several optimisation algorithms will be discussed. Some of the improvements mentioned here are incorporated in those optimisation algorithms. The algorithm TSPspea, which is used with optimisation problems in chapter 5, 6, 7 and 8 also uses some of these improvements (see appendix B).

Chapter 4

Implementations of optimisation algorithms

With the theoretical knowledge on optimisation algorithms gathered in chapter 2 and 3 in mind some implementations of optimisation algorithms will be evaluated. The optimisation algorithms of Matlab can give a feeling for optimisation problems and methods to solve these problems. Through the Internet some existing GAs can be found, which can be altered to fit the optimisation problem at hand.

4.1 Gradient-based optimisation algorithms

The search for a suitable optimisation algorithm was first started by evaluating the optimisation toolbox from Matlab. In this toolbox a multiobjective optimisation algorithm is present as well, which makes use of a gradient based method. An optimisation method should guide the search towards the optimal solution. In case of multiobjective optimisation the method should guide the search towards the Pareto front.

There are several SOO algorithms in the Matlab optimisation toolbox: constrained and unconstrained, linear and nonlinear programming algorithms. These algorithms were tested with the function: $min_{x\in\mathbb{R}} y = a * sin(\frac{x}{b}) + c * x$, to see if they were able to get out of local minima. If a method gets stuck in local minima this means that the true optimal solution will not be found. Some of the methods got stuck in local minima (`FMinBnd, FMinSearch`) and some were able to look a bit further (`FMinCon, FMiniMax, FMinUnc`). The Goal Attainment method of Gembicki[9] implemented in the function `fgoalattain` from this toolbox is a MOO algorithm. In this constrained nonlinear minimisation method the objectives (F_j) are formulated as design goals (F_j^*). The problem formulation allows the objectives to be under- or over-achieved enabling the designer to be relatively imprecise about the initial design goals. The relative degree of under- or over-achievement of the goals is controlled by a vector of weighting coefficients, $w = \{w_1, w_2, ..., w_m\}$, and is expressed as a standard optimisation problem using the following formulation:

$$\text{minimise} \quad \gamma, \text{ with } \Omega \text{ the design space, such that} \quad F_j(x) - w_j\gamma \leq F_j^*, \quad j = 1, .., m$$
$$\gamma \in \mathcal{R}, X \in \Omega$$

It was found that the results of an optimisation with this method are strongly dependant on the initial point. The method can get stuck in the first local optimum it encounters when using a different initial point. So beforehand it is impossible to tell how close the result of the optimisation will be to the global minimum. Therefore, it is not a very good method.

4.2 Implementations of multiobjective GA

The next step is to optimise a multiobjective problem with a genetic algorithm. Thus an implementation of a Multiobjective Genetic Algorithm (MOGA) is needed. Time can be saved by using an existing implementation of a GA, instead of implementing a new algorithm. Alternatively there are many implementations of simple GAs available on the Internet. The problem is to find the implementation that works best for the optimisation problem to be solved. Because suitable implementations of MOGAs are not easy to find on the Internet, an attempt is made to turn a simple GA in a MOGA.

4.2.1 A Genetic algorithm in Matlab

The first choice was trying to extend the SOGA `genetic.m`, implemented in Matlab, to a MOGA. To make `genetic.m` a MOGA the following principles had to be built in the SOGA:

- Ranking: Non-dominated sorting was used as can be found in papers of Fonseca and Flemming [7]. This ranking principle also influences the selection method which takes place in the reproduction part of the algorithm.

- Selection: The roulette-wheel selection was chosen.

- Elitism: The determination of the best individuals of the population so far and the preservation of these individuals in a separate group, the elite group. This method can be found in the PhD thesis of Zitzler ([21]).

- Clustering: A method to select and delete individuals which show little genetic diversity, because they are close to other individuals in the objective space. This method can also be found in [21].

- Distance: A way to determine the distance between two individuals in the objective space.

See chapter 3 for more information on these principles. `Genetic.m` uses a binary representation of the individuals. To calculate the objectives the binary string has to be decoded to floating points.

In [21] some analytical test functions are used. In chapter 5 these functions are listed and in appendix A a background on these functions is given. These functions were used to test the extended version of `genetic.m`. This algorithm gave some unsatisfactory results. It cannot find a uniform distribution of points on the Pareto front. Furthermore it shows different points in the Pareto set with the same value for one objective but a different value for the other objective, see figure 4.1. This is in contradiction with the definition of Pareto optimality, since some of the points from this Pareto front are clearly dominated. An extended version of `genetic.m` might be able to give good results, but the implementation is not flawless. It can be concluded that there is too much risk and effort in extending this SOGA to a MOGA.

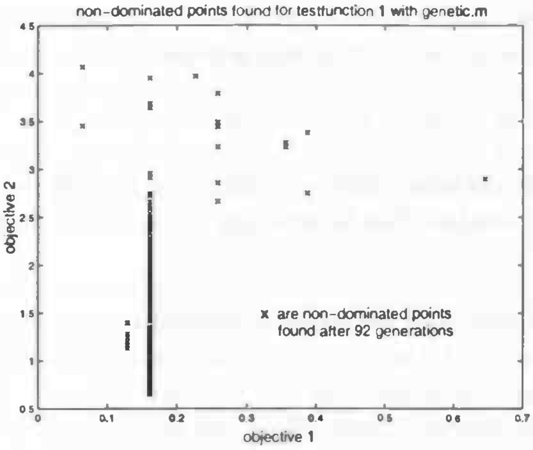


Figure 4.1: The results from test function 1 where x represents the Pareto set found by `genetic.m`

4.2.2 A genetic algorithm library

The next attempt concerned a GA software library developed at MIT [8]. This library, GAlib, is also based on single objective genetic algorithms. The possibilities of changing this SOGA library to a MOGA library were thoroughly investigated. It turned out there is no easy way to preserve the multiple fitness values. Furthermore, because of the structure of this library and the close relation between all functions, all changes made to this library, have effect on many other parts of the library. Extending GAlib to a MOGA library will therefore be very difficult and time consuming.

4.2.3 Strength Pareto Evolutionary Algorithm

In another search on the Internet for a MOGA, a program skeleton for a multiobjective genetic algorithm was found. This MOGA is called Strength Pareto Evolutionary Algorithm (SPEA). With help from E. Zitzler, the author of this algorithm, this skeleton was filled to a MOGA, called TSPspea. This is an algorithm meant to optimise a two-objective traveling salesman problem.

SPEA uses a mixture of established and new techniques in order to approximate the Pareto-optimal set [21]. Similarly to other MOGAs, it

- stores those individuals externally that represent an non-dominated front among all solutions considered so far;
- uses the concept of Pareto dominance in order to assign scalar fitness values to individuals; and
- performs clustering to reduce the number of individuals externally stored without destroying the characteristics of the Pareto front.

SPEA is unique in four respects:

- It combines the above three techniques into a single algorithm;

- The fitness of a population member is determined only from the individuals stored in the external set; whether individuals in the population dominate each other is irrelevant;
- All individuals in the external set participate in selection;
- A new Pareto-based niching method is provided in order to preserve diversity in the population. A niching method like fitness sharing helps to maintain diversity of the genes in the population.

First an initial population and an empty external set are generated. Then the external set ($\bar{P}$) is updated. $\bar{P}$ is filled with the non-dominated points from the current population (P) and points that are no longer non-dominated are removed from $\bar{P}$. A clustering method keeps the number of individuals in this set small. Next the fitness values of the individuals of P and $\bar{P}$ are calculated. This is followed by selection and recombination, after which the external set is again updated to the new current population, and so forth. The algorithm stops when a fixed number of generations, determined by the user, have evolved.

More information on SPEA can be found in [21].

4.3 TSPspea

The TSPspea is an algorithm orientated on a two-objective traveling sales man problem (TSP). To get familiar with the algorithm and to validate the code, first some traveling salesman problems are optimised. The results are listed in chapter 5.

4.3.1 Traveling Salesman Problem

The traveling salesman problem is a single objective optimisation problem (SOP). It is defined by a number l of cities and an $l \times l$ matrix $C = (c_{i,j})$ which gives for each ordered pair (i, j) of cities the distance $c_{i,j}$ from city i to city j. The optimisation goal is to find the shortest route for which each city is entered and left exactly once. This can be turned in a multiobjective problem (MOP) by adding an arbitrary number of distance matrices. Formally, given l cities and a set $\{C_1, C_2, ..., C_k\}$ of $l \times l$ matrices with $C_h = (c^h_{i,j})$, minimise f(π) = $(f_1(\pi), f_2(\pi), ..., .f_k(\pi))$ with

$$f_i(\pi) = (\sum_{j=1}^{l-1} c^i_{\pi(j),\pi(j+1)}) + c^i_{\pi(l),\pi(1)}$$

and where π is a permutation over the set $\{1, ..., l\}$. Altogether, three problem instances were considered with 100, 250 and 500 cities, each having two objectives. The distance matrices were generated at random, with the number of cities as seed. A random number in the interval [1,100] was assigned to each $c^h_{i,j}$.

4.3.2 The algorithm

TSPspea is a MOO that is designed to optimise a TSP which has been made into a multiobjective problem. TSPspea works according to the same principles as SPEA, but because the algorithm is orientated on the TSP some operators are specific for TSPspea. An evolution can be described in 7 steps.

1. Two pseudo random distance matrices of size $N \times N$, with N the number of cities, are generated. It uses N as a seed, so each run with N cities uses the same distance matrices. Because of this the results of different runs with the same number of cities can be compared.

2. A pseudo random initial population of individuals which represent feasible solutions is generated. Each individual is a permutation of all town numbers and represents a route to pass all the cities in both distance matrices exactly once.

3. All the non-dominated points so far are put in the external set called NondominatedSet. The two objectives are calculated for each individual, i.e. the total distances of the route for both distance matrices.

4. For selection at most MaxParetoSize non-dominated individuals from the NondominatedSet are put in the ParetoSet using a clustering method. The set MatingPool is filled with parents selected from the union of the population and the ParetoSet. As selection method binary tournament selection is used. Two individuals from the union set of the ParetoSet and the current generation are selected for a tournament. The winner from this rank-based tournament will become a parent.

5. The new population will be filled with the new children if crossover has occurred or with the original parents from the set MatingPool if crossover did not take place.

6. This new population is subjected to mutation and so the new generation of individuals is formed which can evolve starting again at step 3.

7. The algorithm stops after an user-determined number of generations.

The output consists of a file with the initial population, a file with all individuals of all the generations and a file with the NondominatedSet. The individuals in this set are the non-dominated individuals from all generations.

Chapter 5

Validations of SPEA

5.1 Validation of TSPspea

Before TSPspea can be changed to be used on other optimisation problems the algorithm has to be validated. As test problems the TSP for 100, 250 and 500 cities are optimised. Each problem is optimised several times with different seeds. This seed determines all the pseudo random numbers used in the evolution of the population, except for the generation of the distance matrices where the number of cities is used as a seed. This means the results of runs with different seed can be compared. Running the same problem with several seeds shows whether the problem is very sensitive to the seed. If all seeds render similar Pareto fronts, this is not the case. A problem which is very sensitive to the seed should always be optimised several times with different seeds to make sure the resulting Pareto front lies close to the *real* Pareto front.

The test problems are carried out using the following parameter settings:

Number of generations T	:	500
Population size N	:	equal to l (number of cities)
Crossover rate p_c	:	0.8
Mutation rate p_m	:	0.1

A population size of $\frac{4}{5} \cdot N$ is used with an external set size of $\frac{1}{4} \cdot N$.

5.1.1 Results

Figure 5.1 contains the results for all 3 problems. All optimisations of the three problems result in good Pareto fronts. This means that the results are similar to the results in [21] on the TSPs and thus the TSPspea used is validated. An extra plot of some more Pareto fronts found for the TSP with 250 cities using different seeds, shows that the optimisation of the TSP highly depends on the seed used. This means that TSPs have to be optimised several times with different seed to make sure that the *real* Pareto front, as it can be found in [21], is reached.

This validated TSPspea can be adapted to make it applicable to other kind of problems which require other chromosomes and reproduction operators.

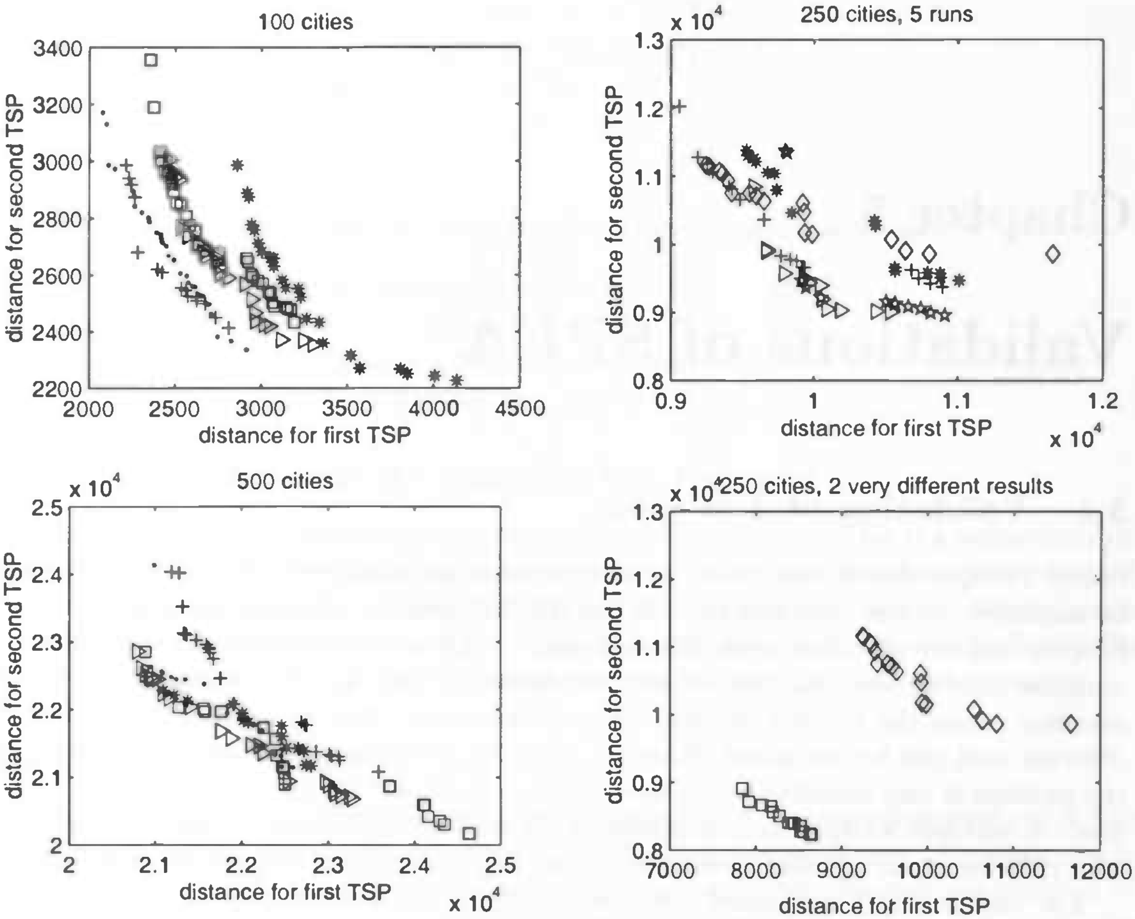


Figure 5.1: TSP for 100, 250 and 500 cities. The different symbols (+, . ,□, *,△) represent the results rendered with different seeds.

5.2 Validation of FPspea

FPspea, floating point Spea, is an altered version of TSPspea. It uses floating point coding for the chromosomes. Six analytical test functions are used to validate FPspea. These test functions are mentioned in [21] and in the work of Deb [4]. Each test function emphasises a different property which could give a MOGA difficulties to properly reach the Pareto front. By using a specific test function as optimisation problem, a MOGA can be tested on its ability to deal with that specific difficulty. Background on these test functions and properties is given in Appendix A. All test functions used are structured in the same manner and consist of three functions f_1, g, h [4]:

$$\begin{array}{lrcl} \text{Minimise} & \mathbf{t}(\mathbf{x}) & = & (f_1(x_1), f_2(\mathbf{x})) \\ \text{subject to} & f_2(\mathbf{x}) & = & g(x_2, ..x_n) \cdot h(f_1(x_1), g(x_2, ..., x_n)) \\ \text{where} & \mathbf{x} & = & (x_1, ..., x_n) \end{array}$$

The function f_1 is a function of the first decision variable only, g is a function of the remaining $n-1$ variables and h is a function of f_1 and g. The test functions differ in these three functions as well as in the number of variables n and in the values the variables may take. For these analytical test functions, the analytical Pareto optimal solutions can be calculated, so the position and shape of the *real* Pareto front is known for these test functions.

Test function $\mathbf{t_1}$ has a convex Pareto-optimal front:

$$\begin{array}{lcl} f_1(x_1) & = & x_1 \\ g(x_2,...x_n) & = & 1+\frac{9}{n-1}\sum_{i=2}^{n} x_i \\ h(f_1,g) & = & 1-\sqrt{\frac{f_1}{g}} \end{array}$$

where $n = 30$ and $x_i \in [0,1]$.

Test function $\mathbf{t_2}$ is the non-convex counterpart to t_1:

$$\begin{array}{lcl} f_1(x_1) & = & x_1 \\ g(x_2,...x_n) & = & 1+\frac{9}{n-1}\sum_{i=2}^{n} x_i \\ h(f_1,g) & = & 1-(\frac{f_1}{g})^2 \end{array}$$

where $n = 30$ and $x_i \in [0,1]$.

Test function $\mathbf{t_3}$ represents the discreteness features: its Pareto-optimal front consists of several non-contiguous convex parts:

$$\begin{array}{lcl} f_1(x_1) & = & x_1 \\ g(x_2,...x_n) & = & 1+\frac{9}{n-1}\sum_{i=2}^{n} x_i \\ h(f_1,g) & = & 1-\sqrt{\frac{f_1}{g}}-\frac{f_1}{g}sin(10\pi f_1) \end{array}$$

where $n = 30$ and $x_i \in [0,1]$.

Test function $\mathbf{t_4}$ contains 21^9 local Pareto-optimal sets and therefore tests for the EAs ability to deal with multi-modality:

$$\begin{array}{lcl} f_1(x_1) & = & x_1 \\ g(x_2,...x_n) & = & 1+10(n-1)+\sum_{i=2}^{n}(x_1^2-10\ cos(4\pi x_i)) \\ h(f_1,g) & = & 1-\sqrt{\frac{f_1}{g}} \end{array}$$

where $n = 10, x_1 \in [0,1]$ and $x_2, ..., x_n \in [-5,5]$.

Test function $\mathbf{t_5}$ describes a deceptive problem and distinguishes itself from the other test functions in that x_i represents a binary string:

$$\begin{array}{lcl} f_1(x_1) & = & 1+u(x_1) \\ g(x_2,...x_n) & = & \sum_{i=2}^{n} v(u(x_i)) \\ h(f_1,g) & = & \frac{1}{f_1} \end{array}$$

where $u(x_i)$ gives the number of ones in the bit vector x_i (unitation),

$$v(u(x_i)) = \begin{cases} 2+u(x_i) & \text{if } u(x_i) < 5 \\ 1 & \text{if } u(x_i) = 5 \end{cases}$$

Test function $\mathbf{t_6}$ includes two difficulties caused by the non-uniformity of the objective space: Firstly, the Pareto-optimal solutions are non-uniformly distributed along the global Pareto front (the front is biased for solutions were $f_1(x_1)$ is near one); secondly, near the Pareto-optimal front the density of the solutions is minimal and away from the front the density of solution is high:

$$
\begin{aligned}
f_1(x_1) &= 1 - e^{-4x_1} sin^6(6\pi x_1) \\
g(x_2, ...x_n) &= 1 + 9(\frac{\sum_{i=2}^{n} x_i}{n-1})^{0.25} \\
h(f_1, g) &= 1 - (\frac{f_1}{g})^2
\end{aligned}
$$

where $n = 10, x_i \in [0, 1]$.

All test problems are carried out using the following parameter settings:

Number of generations T	:	250
Population size N	:	100
Crossover rate p_c (one-point)	:	0.8
Mutation rate p_m (per bit)	:	0.01

5.2.1 Implementation of the analytical test functions

A solution in TSPspea is represented by a permutation chromosome. This is a permutation of all city numbers in the sequence the cities are visited by the salesman. For the optimisation of analytical functions different chromosomes are needed and therefore the recombination operators, which work on chromosome level, have to be adjusted to these new chromosomes. Two different chromosome types are used: the floating point chromosome and the binary chromosome.

Floating point chromosome

For optimisation problems with objectives that are functions of floating point numbers, a floating point chromosome is needed. The floating point chromosome needs recombination operators different from operators for the permutation chromosome. This chromosome consists of an array of floating points. With mutation and crossover the sequence of the alleles is no longer of importance. The crossover operator now simply selects a crossover site and performs a one-point crossover on that point between the two parents. The mutation selects a mutation site and gives the concerning allele a new floating point from the correct interval. P_m represents the probability of mutation to occur per individual. p_m, as mentioned earlier in this section, represents the probability of mutation to occur per allele of a arbitrary chromosome. With p_m fixed, P_m depends on the length of the chromosome and the value of p_m.

Binary chromosome

The objectives of test function 5 are functions of binary input. Thus test function 5 requires individuals represented by a binary chromosome. Some other optimisation problems, which have objectives that are functions of floating points, still evolve better using binary chromosomes as representation of the solutions. The reason for this could be that binary recombination operators work better on this specific problem. To optimise these functions BitSpea, a SPEA based on binary chromosomes, is needed. This SPEA also needs adjusted bit-level recombination operators to operate on the binary chromosome.

Furthermore a decode function is needed to translate a chromosome consisting of bits into a chromosome with floating point alleles for the test functions that have objectives that need floating points as input. The recombination operators still work on the binary chromosome with this test functions. Next to the probability of mutation to occur per individual P_m, with binary coding there is also $P_{m_{bit}}$, which represents the probability of mutation to occur per allele of a chromosome that is already selected to be mutated. Unless mentioned otherwise $P_{m_{bit}}$ is set to 0.01, just as p_m.

This decode function takes per variable a fixed number of binary alleles which represent the range of this variable. These sets of binary alleles can be seen as the genes of the chromosome.

5.2.2 Results

Per test function the results are discussed. The results are compared with the results in [21]. These results give a hint to how close FPspea can get to the *real* Pareto front. In [21] the same test functions are used with a SPEA that most likely differs from FPspea. Therefore, the results in [21] should be similar but do not have to be exactly the same.

Test function 1

Floating point and binary coding give the same results and they are similar to the results shown in [21] and to the *real* front (figure 5.2).

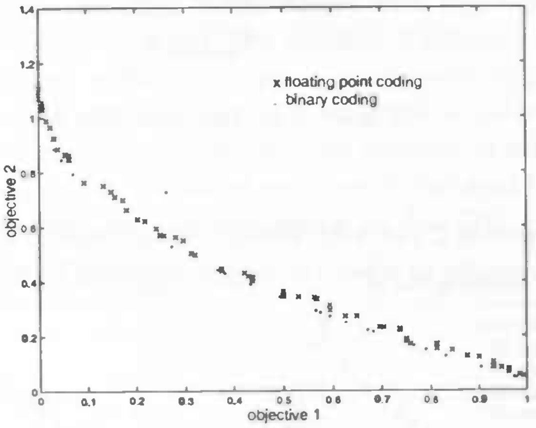


Figure 5.2: The Pareto front of test function 1.

Test function 2

- **Floating point:** With this test function the influence of mutation on the performance of FPspea is revealed.

  When P_m is set to 0.1, it takes FPspea 600 generations to get near the Pareto front instead of the 250 generations used in [21] (figure 5.3). P_m set to 0.01 gives worse results. This means that mutation is indeed a necessary part of the genetic algorithm. Next is an optimisation with more-point mutation. This different type of mutation mutates 4 alleles at random positions each time mutation occurs. This optimisation does not give good results (as shown in figure 5.3), which is a sign that much mutation will not speed up the convergence towards the Pareto front. When P_m is set to 0.3 (i.e. $P_{m_{bit}} = 0.01$ by approximation) the optimisation gives the same results as can be found in [21] and lies very close to the *real* front (figure 5.4a). With P_m set to 0.5, the results do not improve much (figure 5.3).

Conclusion: Mutation speeds up the algorithm. With a lower value for P_m it takes more generations to get the same results. Too much mutation, as used in more-point mutation, will lead to pure random search and thus will slow down the algorithm. Therefore P_m should not be set too high. P_m set to 0.3 gives the best results. A more extensive study on the influence of P_m and other parameters of a genetic algorithm is presented in chapter 6.

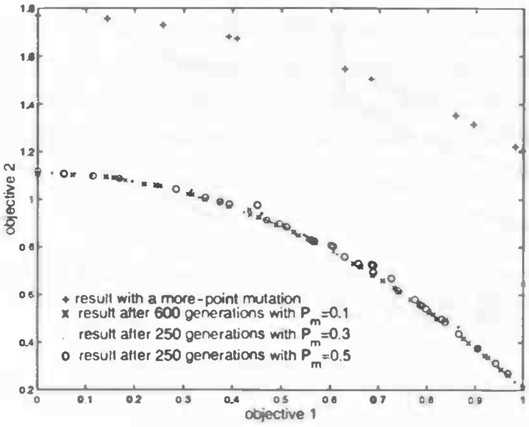
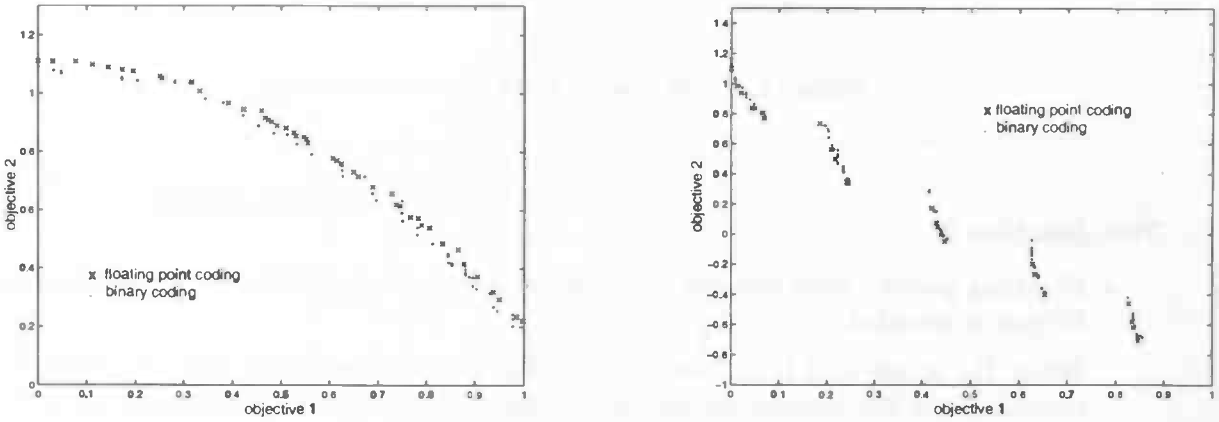


Figure 5.3: Test function 2.

- **Binary:** Test function 2 ran with binary representation of the chromosome gives almost exactly the same results as when the test is ran with floating point coding with P_m set to 0.3 (figure 5.4a).



(a) The results from function 2.

(b) The results from function 3.

Figure 5.4: Test function 2 (a) and 3 (b).

Test function 3

Floating point and binary coding give the same results, which are similar to the results shown in [21] and to the *real* front of this function. With this function P_m was set to 0.3, the value that turned out to work best for test function 2. This value also results in a good optimisation of test function 3 (figure 5.4b). The resulting Pareto front looks a bit different than the Pareto fronts of the previous optimised test functions. This is caused by the discrete features of this test function, as is mentioned at the begining of section 5.2.

Test function 4

- **Floating point:** The results of this test function look just like the *real* Pareto-front(figure 5.5a). The results in [21] lie further away from the *real* front. The differences in the results can be explained by possible small differences between the algorithms used, originated, for example, by using different types of coding of the individuals. The Pareto front found seems totally independent of the seed used, because each run with different seeds results in a Pareto front which is in the same location. This total independence of the seed is a sign that this optimisation problem might suffice with significant less generations to get close to the *real* Pareto front. Lowering the number of generations shows that for this test function 20 generations are sufficient to get a good result (figure 5.5b). However, with only 20 generations in the evolution the evolution is still sensitive to the seed (figure 5.5d). When using an evolution of 30 generations this problem is solved and the optimisation results in a Pareto front that is no longer sensitive to the seed (figure 5.5c). This is still a much shorter evolution than the 250 generations used in [21].

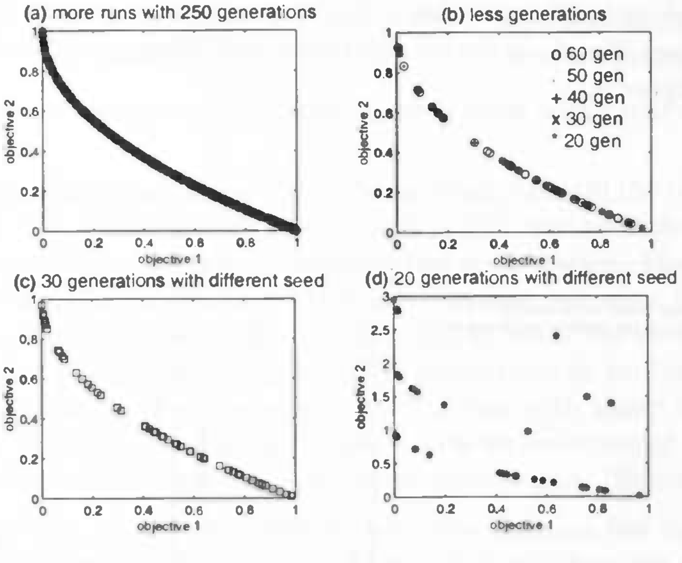


Figure 5.5: Test function 4, floating point coding.

- **Binary:** The results from the optimisation with a binary chromosome do not directly lead to the desired Pareto front. Varying P_m and $P_{m_{bit}}$ reveals no results from which any logic can be derived (figure 5.6a). This is caused by the high sensitivity of this problem to the seed used as shown in figure 5.6b. With these optimisations P_m is set to 1 and $P_{m_{bit}}$ is set

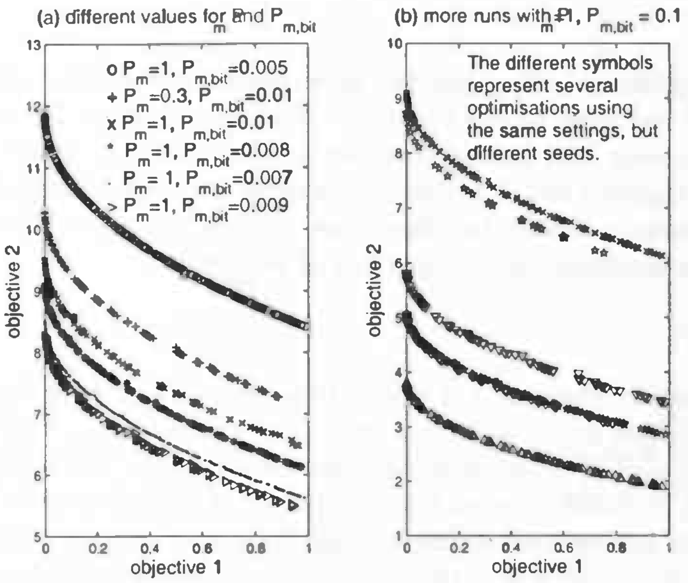
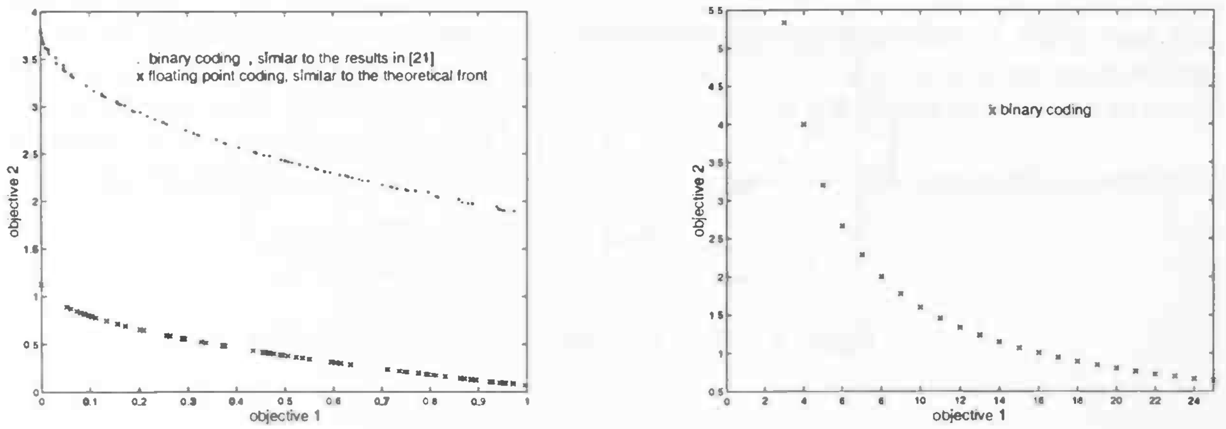


Figure 5.6: Test function 4, binary coding.

to 0.01, the same values as used in [21], but the seeds differ per optimisation. The resulting fronts differ greatly from each other. Among the resulting fronts is a front that is very similar to the one shown in [21]. This means that this test function has to be optimised several times with different seeds to find a Pareto front close to the *real* Pareto front. The best fronts resulting from the optimisations with binary and floating point coding are both plotted in figure 5.7a.



(a) Test function 4.

(b) Test function 5, binary coding.

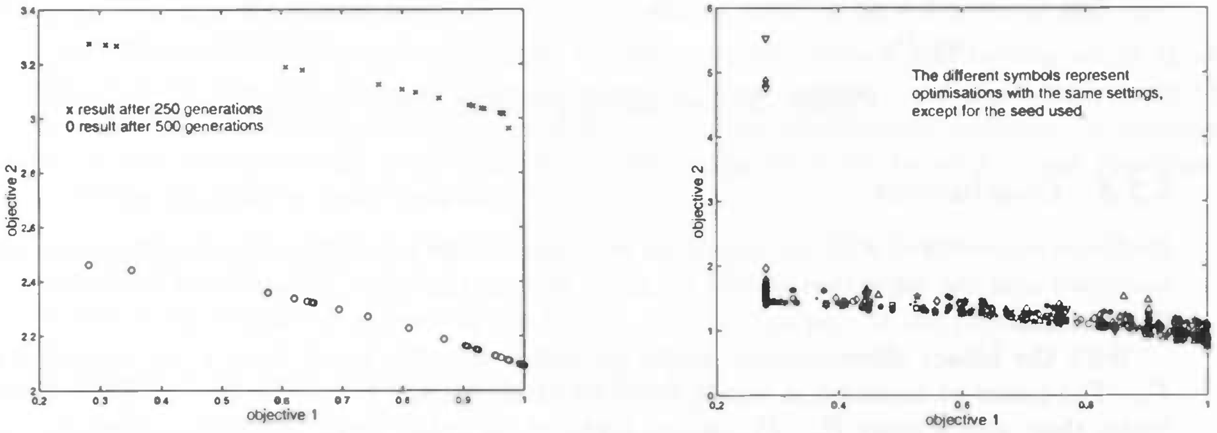
Figure 5.7: Test function 4 and 5.

Test function 5

- **Binary:** This test function is only optimised with a binary chromosome because it has binary objectives. The results were satisfactory and look just like the *real* front (figure 5.7b).

Test function 6

- **Floating point:** After 250 generations the Pareto front found by FPspea is not even close to the *real* Pareto front and thus gives results worse than shown in [21]. After 500 generations the resulting front is significantly better, so more generations will most likely result in an even better front (figure 5.8a).



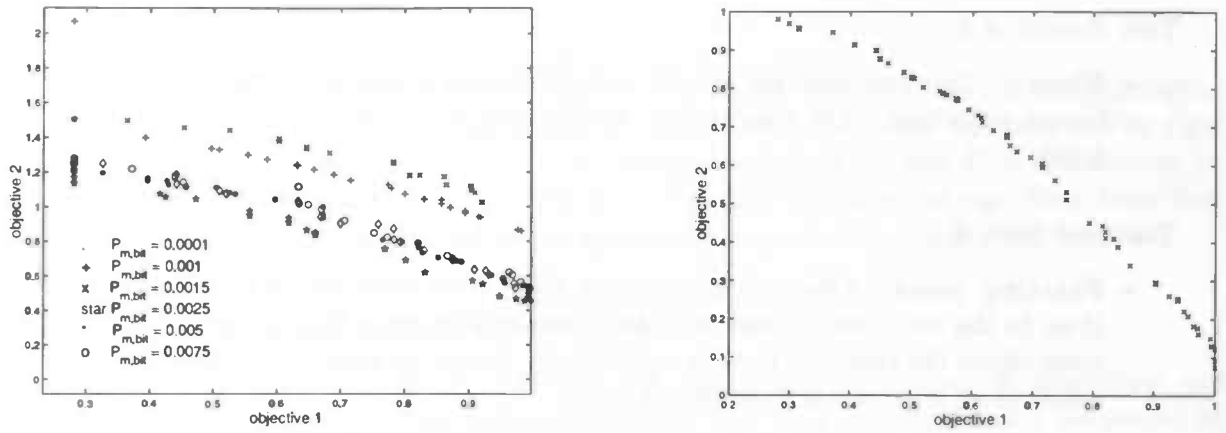
(a) Test function 6 with above 250 generations and below 500 generations.

(b) Many runs with different seeds of test function 6.

Figure 5.8: Test function 6 with floating point (a) and with binary coding (b).

- **Binary:** Binary coding results in a better Pareto front, but still the results are not as good as the results in [21]. The *real* front has one end in (0,1) and the other end in (1,0), but it has about the same shape as the front found in this optimisation. More runs with different seeds yield about the same results, so this function does not seem to be very dependent on its initial population (figure 5.8b). An attempt to reach the same Pareto front found in [21] is made by giving the population 1500 generations to evolve. BitSpea now finds a Pareto-front similar to the front shown in [21], but with many more Pareto optimal solutions on it. This is to be expected because with an evolution of 1500 generations the set of non-dominated individuals has a lot more time to grow (figure 5.9a).

  The influence of $P_{m_{bit}}$ is not very obvious with this function but figure 5.9a shows that $P_{m_{bit}}$ set to 0.0025 gives the best results. This setting still does not give the same results as shown in [21]. It must be concluded that FPspea can not optimise test function 6 to the fullest with the given parameter settings. With more generations in the evolution however the optimisation does get close the *real* Pareto front with both binary and floating point coding (see figure 5.9b).



(a) Test function 6 with different values for $P_{m_{bit}}$.

(b) Test function 6 with 1500 generations and binary coding.

Figure 5.9: Test function 6 using binary coding.

5.2.3 Conclusions

Problems encountered with the algorithm were small. The population size should not be chosen too small and the algorithm should be given enough time (i.e. generations) to evolve to the Pareto front.

With the binary chromosomes, initial problems were the result from a low mutation rate P_m. The power of mutation is hereby demonstrated, because with a higher P_m, the results are better then with a lower P_m. P_m set too high, on the other hand, will lead to random search, so some experiments may be needed do determine the correct value for P_m. The results can differ significantly dependent on the coding used for the chromosome. This shows that with each optimisation problem several codings should be tested, to see which works best with this specific problem.

Most test functions are highly dependant on the initial population. Because the initial population is determined by the seed used, an optimisation should always be run several times with different seeds.

The optimisation of test function 6 with FPspea is the only optimisation which does not give a totally satisfying result. The optimisation of test function 4 with floating point coding however gives a much better results then shown in [21]. These differences may be due to small differences in the algorithms used.

Overall, the genetic algorithm works well and can be adapted for other optimisation problems.

Chapter 6

Strategies for genetic algorithms

Chapter 3 shows that a genetic algorithm has many parameters. When looking at all these possibilities the question arises whether there may be some general strategies with respect to some of these parameters applicable to the different types of optimisation problems. To examine this the analytical functions 1 and 5 as mentioned in chapter 5 will be used as test functions. The parameters used in these tests are:

- population size (*pop*),
- size of the Pareto set as used in the SPEA, expressed as part of the population size (*ps*),
- chance of the event of mutation to take place (P_m),
- chance of the event of crossover to take place (P_c),
- the number of generations in an evolution (*gen*).

The influences of the variables P_m and *gen* on some of the test functions are already shown in chapter 5.

Both test functions are optimised with the following values for the parameters:

pop	10, 100, 200
ps	1/10, 1/4, 2/3
P_m	0.1 and 0.5
P_c	0.5 and 0.8

The choice to use either binary or floating point coding is based on the results found in chapter 5. The coding which suits the specific test function best in that chapter is used.

The results of all the tests are judged for the influences of the parameters in terms of:

- the time needed for SPEA to evolve,
- the distribution of solutions found on the Pareto front,
- the distance of the found Pareto front to the *real* Pareto front.

These considerations to determine the strategies for the test functions do not suite all practical applications. Different problems need different strategies, but the test results shown in this chapter can be used to determine a strategy for the optimisation problem at hand.

6.1 Results

With the mutation and crossover parameters P_m and P_c set to specific values, the population size, Pareto size and number of generations are varied with both test functions. The right combination of *gen*, *ps* and *pop* needed for the different values of P_m and P_c is determined, by first determining the best values for *gen* and *ps* for the different values of *pop*. From these four settings the best achieving is selected as the advised strategy for the specific test function.

6.1.1 Test function 1, using floating point coding

The computation time needed for the different evolutions differs per number of generations, size of the populations and slightly per size of the Pareto sets. It is not significantly influenced by the settings of P_m and P_c. With different settings on SGI-O2-R5000 workstations the evolutions took approximately:

pop	*gen*	time (sec.)
10	10	1
10	100	2
10	1000	20
100	10	2
100	100	30
100	1000	360
200	10	3
200	100	40
200	1000	1140

The results are shown in figures 6.1 and 6.2. When the different Pareto set sizes do not influence the reached fronts, the fronts found with the different sizes for the Pareto set cover each other.

The conclusion for the optimisations with different settings for P_m and P_c are the same: The evolution needs 1000 generations for all three different population sizes to get close enough to the *real* Pareto front. The evolutions with *pop* = 100 result in Pareto fronts with a lot more non-dominated points then the evolutions with *pop* = 10. The Pareto fronts of an evolution with *pop* set to 200 and 100 differ in the same manner. The different sizes for the Pareto set do not influence the results a great deal.

For all settings of P_m and P_c the fronts that are found after 1000 generations with *ps* set to 1/4 of *pop* for the 3 different population sizes are plotted in the last plot of the corresponding figures (6.1 a and b and 6.2 a and b).

Looking at these last plots, the followings remarks can be made:

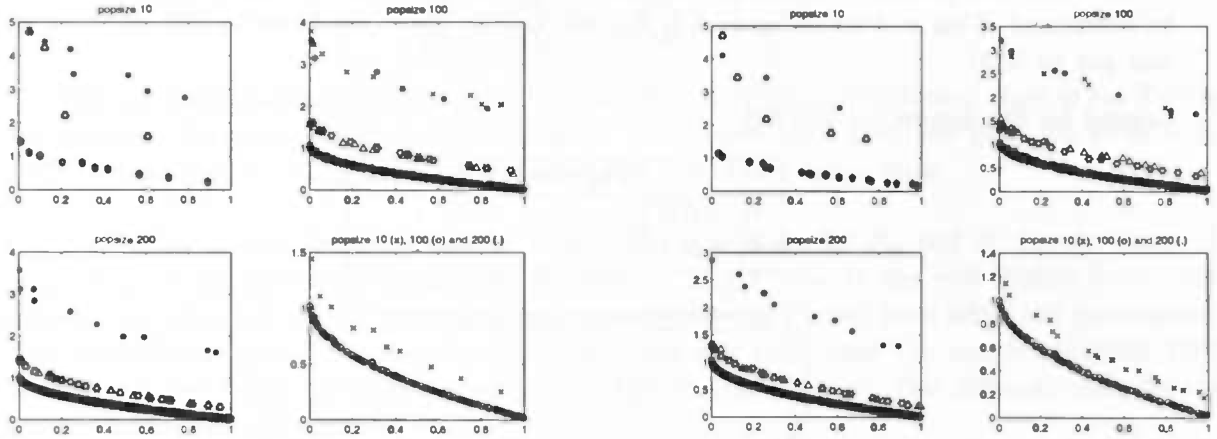
- With P_m set to 0.1 the same conclusions apply to both P_c set to 0.5 and 0.8 (figure 6.1 a and b): There is no difference visible between the fronts reached with *pop* = 100 and *pop* = 200 . However, the time needed for an evolution of 1000 generations with *pop* = 100 takes about 6 minutes. The same evolution with *pop* = 200 takes about 19 minutes. The evolution with *pop* = 10 results in a front that is less good than the fronts of the other two optimisations.

- With P_m set to 0.5 the same conclusions apply to both P_c set to 0.5 and 0.8 (figure 6.2 a and b): The results from the optimisation with *pop* = 200 are only slightly better than the results of the evolution with *pop* = 100, but this evolution takes up much more time. The evolution with *pop* = 10 results in a Pareto front that lays slightly higher than the other two Pareto front, but this may be judged to be close enough to the optimal solutions. The evolution of this population for 1000 generations only takes half a minute.

Conclusion: Test function 1 needs the following settings for all values of P_m and P_c: *pop* = 100, *ps* = 1/4 and *gen* = 1000. But if time is limited a population size of 10 with a Pareto set size of 2 will give satisfactory results when P_m is set to 0.5.

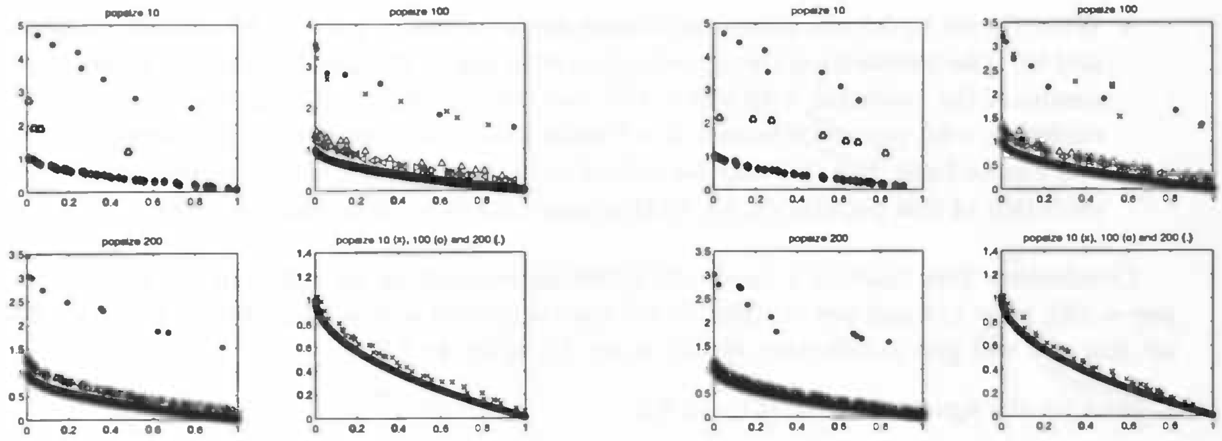
Legend for the figures 6.1, 6.2, 6.4 and 6.5:

ps	1/10 of *pop*	1/4 of *pop*	2/3 of *pop*
gen = 10	×	+	*
gen = 100	△	◁	▷
gen = 1000	□	◇	★



(a) P_m set to 0.1 and P_c set to 0.5 (b) P_m set to 0.1 and P_c set to 0.8

Figure 6.1: Test function 1



(a) P_m set to 0.5 and P_c set to 0.5

(b) P_m set to 0.5 and P_c set to 0.8

Figure 6.2: Test function 1

Now the best results from the optimisation with the different settings of P_m and P_c have to be compared. This is done in figure 6.3. For all optimisations *pop* is set to 100, *ps* is set to 1/4 and *gen* to 1000.

Legend for the figures 6.3 and 6.6:

	$P_m = 0.1$	$P_m = 0.5$
$P_c = 0.5$	×	*
$P_c = 0.8$	○	·

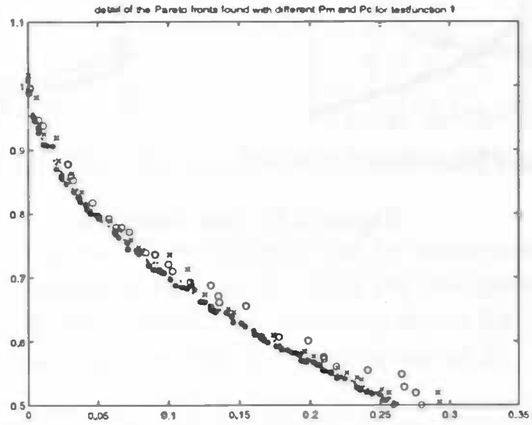


Figure 6.3: Test function 1 with different values for P_m and P_c

P_m set to 0.5 leads to better results than P_m set to 0.1. The value off P_c is not of great influence on the results. With P_m set to 0.1 however, P_c set to 0.8 gives better results, so this setting is chosen. As mentioned earlier, with P_m set to 0.5 a smaller population size ($pop = 10$) can be sufficient. This makes the evolution much faster.

The strategy for test function 1 is: P_m set to 0.5, P_c set to 0.8, *pop* set to 100, *ps* set to 1/4 and *gen* set to 1000. It is remarked that with *pop* set to 10 and *ps* set to 1/4 the evolution is much faster and the results can still be sufficient.

6.1.2 Test function 5, using binary coding

In BitSpea the chance on mutation per bit (pm) is set to 0.01. The computation time needed for the evolutions with different settings on SGI-O2-R5000 workstations is approximated by:

pop	*gen*	time (sec.)	*ps*
10	10	1	
10	100	2	
10	1000	30	
100	10	2	
100	100	30	
100	1000	300	
200	10	5	
200	100	50	
200	1000	500	1/10
200	1000	850	1/25
200	1000	1300	2/3

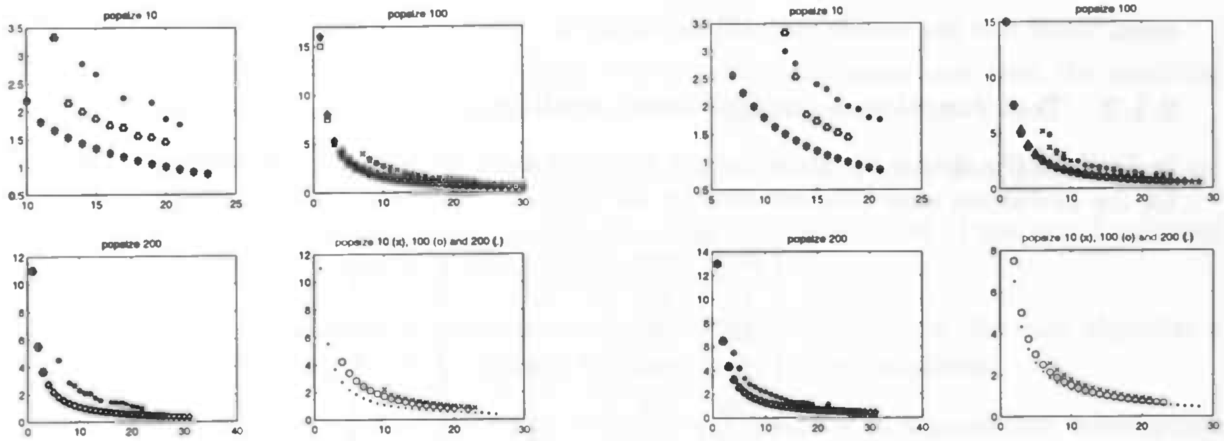
With an optimisation with population size of 200 individuals the different sizes of the Pareto set influence the time needed for the evolution. The larger the Pareto set gets, the more individuals there are to be compared each generation. This takes extra time.

The conclusion for the optimisations with different settings for P_m and P_c are the same: A population of 10 individuals needs 1000 generations to get close to the *real* Pareto front. The populations with 100 or 200 individuals get close to the *real* Pareto front after 100 generations. The front found after 100 generations is just a bit less wide than the one found after 1000 generations but it lies in the same position in the objective space. The different sizes for the Pareto set hardly influence the results.

For all settings of P_m and P_c, for all three population sizes the best fronts are plotted in the last plot of the appropriate figures (6.4 a and b and 6.5 a and b). All fronts are reached using a Pareto size set to 1/4 of the population size. The population of 10 individuals is given 1000 generations to evolve, the other two population sizes 100 generations.

With all settings of P_m and P_c the same conclusions apply: The resulting fronts are almost equally positioned. The Pareto front resulting from the optimisation of 200 individuals is a bit wider than the fronts resulting from the other two optimisations. The front resulting from the optimisation of 100 individuals results in a front that is wider than the front resulting from the optimisation of 10 individuals. The optimisation with 200 individuals tcd Pmakes significant longer than the other two optimisations.

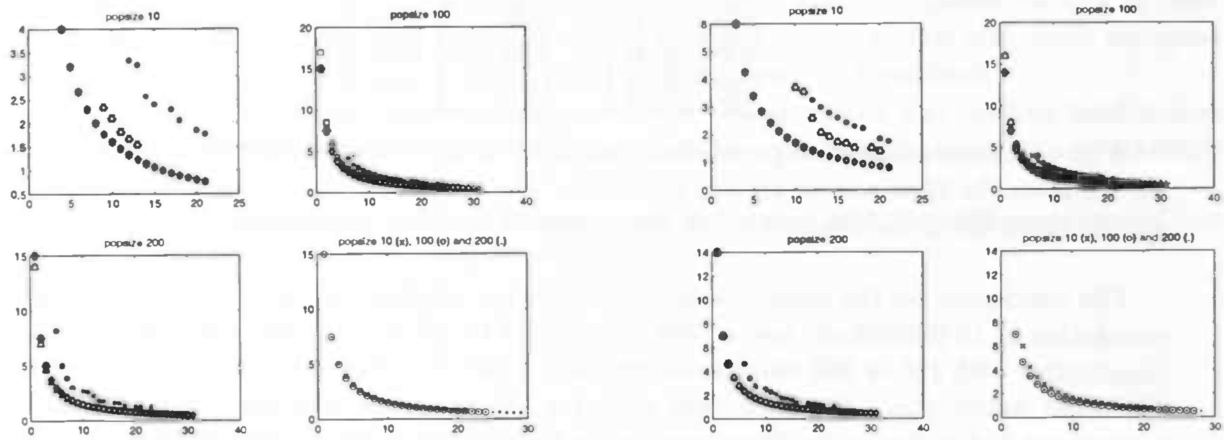
Conclusion: Test function 5 needs the following settings:
pop = 100, *ps* = 1/4 and *gen* = 100, for all values of P_m and P_c.



(a) P_m set to 0.1 and P_c set to 0.5

(b) P_m set to 0.1 and P_c set to 0.8

Figure 6.4: Pareto fronts found for test function 5 with different values of P_m and P_c



(a) P_m set to 0.5 and P_c set to 0.5

(b) P_m set to 0.5 and P_c set to 0.8

Figure 6.5: Pareto fronts found for test function 5 with different values of P_m and P_c

Now the best values for P_m and P_c are determined by comparing the results from the optimisations with the different values of P_m and P_c. This are the optimisations with *pop* = 100, *ps* = 1/4 and *gen* = 100 and the Pareto fronts are shown in figure 6.6. P_m set to 0.5 gives the best results. The optimisation gives equal results for both values of P_c.

The strategy for test function 5 is: P_m set to 0.5, P_c set to 0.5 or 0.8, *pop* set to 100, *ps* set to 25 and *gen* set to 100.

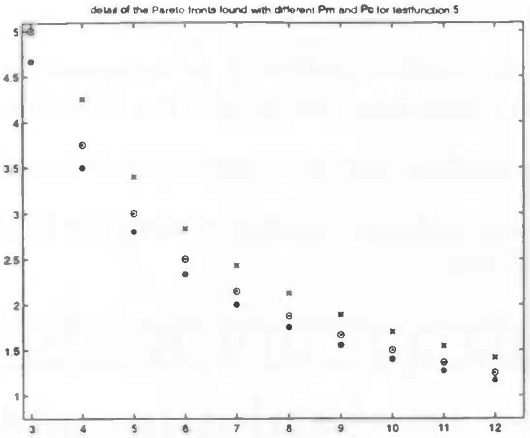


Figure 6.6: Test function 5 with different values for P_m and P_c

6.2 Another influential part of a genetic algorithm

When a genetic algorithm is used to solve an optimisation problem an appropriate chromosome has to be designed. This is certainly not the most straightforward part of using a genetic algorithm. The type of alphabet that will be used, binary or floating point etc., has to be determined. Furthermore, it has to be decided what information has to be put in the chromosome. The length of the chromosome has effect on the result of the optimisation. This is seen with the optimisation problem dealt with in chapter 7, were the problem is optimised several times using different chromosomes. From the results of these optimisations conclusions can be made on this issue.

6.3 Conclusions

It turns out that it is possible to define strategies for both test functions. These strategies are not the same, as was expected. Some parameters, such as population size, have much more influence on the results than other parameters, such as the size of the Pareto set. A relationship between the population size and the number of generations becomes visible. A large population will need more than 10 generations to evolve properly. The time needed for the evolution depends very clearly on the population size and the number of generations used. The strategy for the other parameters is not very dependent on the settings used for P_m and P_c.

6.4 Further investigations

GAs are still under development. There are many things yet to investigate, among which the working of the GA in MOO problems.

The tests run in this chapter show that general strategies can be formed. The strategies can be extended using more parameters on more test functions.

6.4.1 Recommendations

With a multiobjective optimisation problem it is important to make sure that the resulting non-dominated front, the Pareto front, meets with the following properties:

1. The distance of the resulting non-dominated front to the *real* Pareto front should be small.
2. A good (in most cases uniform) distribution along the Pareto front of the non-dominated points found is desirable.
3. The spread of the obtained non-dominated front should be large, i.e., for each objective a wide range of values should be covered by the non-dominated solutions.

This means that genetic drift should be stopped and methods to improve the convergence rate should be found without losing the global search aspect of the genetic algorithm.

For this purpose some mechanisms, as mentioned in chapter 3 and [13] could be implemented in a genetic algorithm. The most important mechanisms are: fitness sharing, elitism, hybrid methods and deme forming. The influences of the different types of implementations of these mechanisms for the different test functions should be investigated. In this way, more extensive strategies for the different types of optimisation problems may be formulated.

Furthermore, the different recombination and selection operators and ranking mechanisms could be tested for codings and test functions. This may also lead to some general strategic conclusions.

To get a complete picture of which strategies are to be used with the different types of test functions, more types of test functions should be tested.

Chapter 7

Design optimisation for the Runge-Kutta method

In this chapter the coefficients of a Runge-Kutta method are optimised using SPEA.

7.1 Introduction to the problem

At National Aerospace Laboratory (NLR) the flow solver Hexadap was developed. Hexadap solves 3D Euler equations on hexahedron grids using a Discontinuous Galerkin (DG) finite element method (for more information on this method see [19]). This method can be considered as a mixture of an upwind finite volume method and a finite element method [12].

To numerically solve the 1-dimensional convection equation, $\frac{\delta u}{\delta t} + c\frac{\delta u}{\delta x} = 0$, it has to be discretised in time and space. DG discretisation makes use of the average value of the solution in the cells of the grid ($\bar{u}$) as well as a linear variation of this value ($\hat{u}$). So the solution in a specific cell can be written as a vector $\left(\begin{array}{c} \bar{u} \\ \hat{u} \end{array} \right)$. Given a wave $w_\theta(x) = e^{i\theta x/h}$ $(0 \leq \theta \leq \pi)$, it can be discretised on a uniform mesh with mesh width h. Let $w_{\theta,h}$ denote the discretised wave. Define the 2×2 matrix $L_h(\theta)$ by $L_h w_{\theta,h} = L_h(\theta) w_{\theta,h}$, with $L = c\frac{\delta u}{\delta x}$ and L_h is the discretised L. The eigenvalues of $L_h(\theta)$ determine the stability properties of the discretisation. $L_h(\theta)$ is influenced by the CFL-number $(\frac{c\delta t}{h})$. The differential equation to be solved is the following:

$$y' = L_h y, \; y \in \mathbb{R}^2, \tag{7.1}$$

with L_h the differential operator. The time discretisation of 7.1 is done with a Runge-Kutta (RK) method. For RK5 as used in Hexadap L_h is split up in a convective and a diffusive part, $L_h = L_h^C + L_h^D$:

$$y_{n+1} = V_5 \cdot y^n, V_5 \in \mathbb{R}^2 \times \mathbb{R}^2 \tag{7.2}$$

With V_5 recursively defined by:

$$\begin{cases} V_0 = I \\ V_i = I - \alpha_i \cdot CFL((L_h^C + \beta_i L_h^D)V_{i-1} + (1 - \beta_i)L_h^D) \end{cases} \tag{7.3}$$

V_5 will be reffered to as the RK-matrix. The RK-matrix, like L_h, depends on θ. The parameters β_i in 7.3 determine how the artificial dissipation of the numerical sheme is calculated. With $\beta_i = 1$ the dissipation at stage i is recalculated and with $\beta = 0$ the dissipation of the previous stage $(i-1)$ is used. For other values of β the dissipation is a weighted average of the dissipation of the previous and present stage.

7.1.1 Objectives

The purpose of this optimisation is to improve the rate of convergence of the solving of the Euler equations. Two means are used for this goal:
Damping: In general holds that the more damping the faster irregularities disappear from the solution. High damping can be generated by minimising

$$\int_{\frac{1}{2}\pi}^{\pi} (|\lambda_1(\theta)| + |\lambda_2(\theta)|)\, d\theta, \text{ where } \lambda_i \text{ are the eigenvalues of the RK-matrix } V_5. \tag{7.4}$$

The absolute value of the eigenvalues of an RK-matrix are plotted in figure 7.1 for $\theta \in [0, \pi]$.

CFL-number: The larger the CFL-number the larger time steps can be taken.

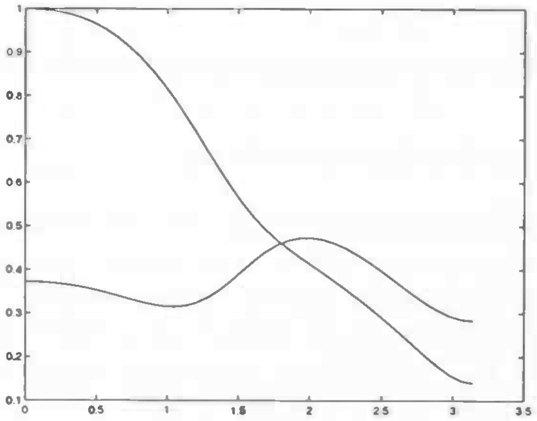


Figure 7.1: The eigenvalues for a RK5 time-integration

The multiobjective optimisation problem involved with Hexadap is the following: at the same time

- minimise integral 7.4 and
- maximise the CFL-number.

Do this under the constraint

- $|\lambda_{1,2}(\theta)| \leq 1 \quad \forall\, \theta \in [0, \pi]$.

7.2 Implementation

To be able to deal with this problem, some changes to the implementation of SPEA are necessary. The chromosome needed for this optimisation problem consists of ten floating points, representing the CFL-number and the nine coefficients for the Runge-Kutta method, $\alpha_1..\alpha_5$ and $\beta_1..\beta_4$. The CFL number has a range of [0.8,2.8], unless mentioned otherwise, the α's and β's lie in the interval [0,1], except α_5 which is always kept at one.

To be able the calculate the Runge-Kutta matrices, some matrix calculations like addition and multiplication of matrices had to be built into FPspea.

7.3 Results

The values for α_i can easily be changed in the Runge-Kutta scheme in Hexadap after an optimisation to see the results, but it is not easy to change the values of the β's. Therefore the optimisation problem will be approached in two ways.

1. Keep the β's fixed at one, which is the value the β's currently have in Hexadap. This approach gives the opportunity to easily show the improvements of a new scheme to Hexadap. With the β's fixed at one, these 'variables' can be put in the chromosome, or they could be left out altogether. This last option shortens the length of the chromosome and will influence the recombination operators.

2. Let the β's vary between 0 and 1, just like $\alpha_1..\alpha_4$.

The results coming from this different optimisation approaches are discussed in this section. The parameter settings for all runs were:

Crossover rate p_c	:	0.8
Mutation rate p_m	:	0.1
ParetoSet of size	:	$\frac{1}{4}$ Population size

The other parameters differ per optimisation and are mentioned in the text.

7.3.1 β's fixed at one.

The optimisation is started with the β's fixed at one but still using them as alleles in the chromosome. The results of three optimisation runs with different parameter settings are shown in figure 7.2. On the x-axis the value of the integral 7.4 is plotted and on the y-axis minus the CFL-number is plotted.

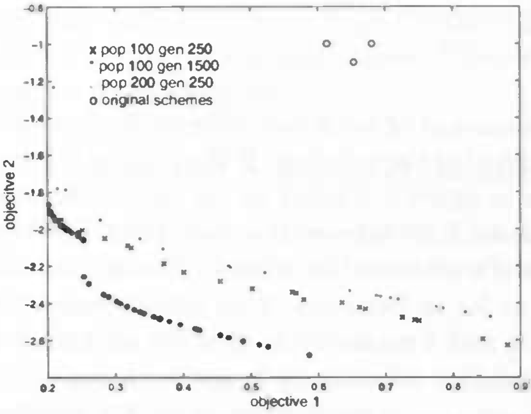


Figure 7.2: Keeping the β's in the chromosome and fixed at one, for 3 different optimisations

To see how close the front of non-dominated points resulting an evolution of 100 individuals for 100 generations (x) lies to the *real* Pareto front this population has also been evolved for 1500 generations (*). The non-dominated points found by the last optimisation lie a bit lower, which means that to get close the *real* Pareto front, more than 250 generations are necessary.

To investigate the influences of the population size, a population of 200 individuals has been evolved for 100 generations (.). The population size does not influence the results a lot, as can be seen in this same plot.

In Hexadap three different sets of parameters are already used, namely the following chromosomes: $[1, \frac{1}{5}, \frac{1}{4}, \frac{1}{3} \cdot \frac{1}{2}, 1, 1, 1, 1, 1]$, $[1, \frac{1}{4}, \frac{1}{6}, \frac{3}{8}, \frac{1}{2}, 1, 1, 1, 1, 1]$ and $[1.1, \frac{1}{4}, \frac{1}{6}, \frac{3}{8}, \frac{1}{2}, 1, 1, 1, 1, 1]$. The fitness values for these sets have been calculated and are plotted with (o) in the same plot 7.2. It is obvious that the individuals found by the GA are much better with respect to the objectives then the old sets of parameters used in Hexadap. Three solutions from these resulting Pareto fronts, are tested in Hexadap. One solution with a very high CFL-number, one solution with a very small value for integral 7.4 and one solution with average values for both objectives. All three perform great with respect to the goal of this optimisation, improving the convergence rate of the solving of the convection equation.

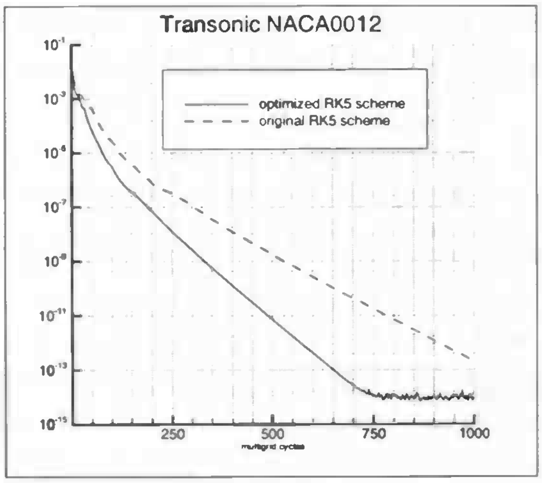
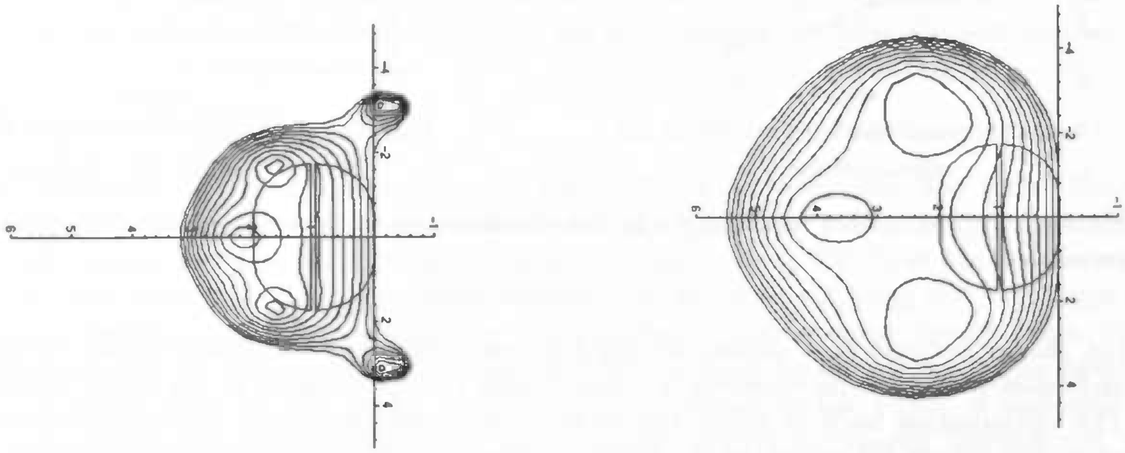


Figure 7.3: Comparing the results of the different scheme's in Hexadap, with on the y axis the residual of the solution.

Figure 7.3 shows the amount of work two different Runge-Kutta schemes need to do to get to a specific size of the residual of the solution. If the residue is zero, the solution is stationary. The optimised RK5 scheme in figure 7.3 refers to the optimised design with CFL-number 2.39069. For α_1 ... α_5 the optimised RK5 scheme has the values 0.0791451 0.163551 0.283663 0.500858 and 1 respectively. The β's all have the value 1. The original scheme in figure 7.3 refers to the scheme as it was used so far in Hexadap. This scheme has a CFL-number of 1.1 and α_1 ... α_5 have the values $\frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}$ and 1 respectively. The β's all have value 1.

The additional information revealed by figure 7.3 helps to choose one specific solution from the Pareto front as the solution of the optimisation. A reduction of four orders in the residuals is desirable, this means a residue level of approximately 10^{-7} should be reached. The optimised scheme needs 185 cycles to get to this level, the original scheme needs 300 cycles. A cycle is a working-unit so a speed up of the convergence rate of 300/185 is accomplished with the optimised scheme. The original scheme in Hexadap has been replaced by the optimised scheme. The design with very large CFL-number is selected because as it turns out the large CFL-number helps better to improve the convergence rate then the improved damping with Hexadap. This is probably caused by the multi-grid algorithm used in Hexadap.

This selection of a solution from the Pareto front is an example of how knowledge on the shape of the Pareto front can help in deciding which solution is the ultimate design. Before optimising it was not known how large the CFL-number could get while keeping a good fitness value for the first objective.



Stability region with the old scheme.

Stability region with the new scheme.

Figure 7.4: Optimisation without β's in the chromosome.

In figure 7.4 the stability regions of both schemes are presented. This demonstrates the improvements made by the new scheme. The 'circles' drawn in the stability regions are the eigenvalues of the RK-matrix with CFL set to one. With the new scheme a larger CFL-number can be used with the eigenvalues still in the stability region. This means that a larger time-step can be used and still the solution remains stable.

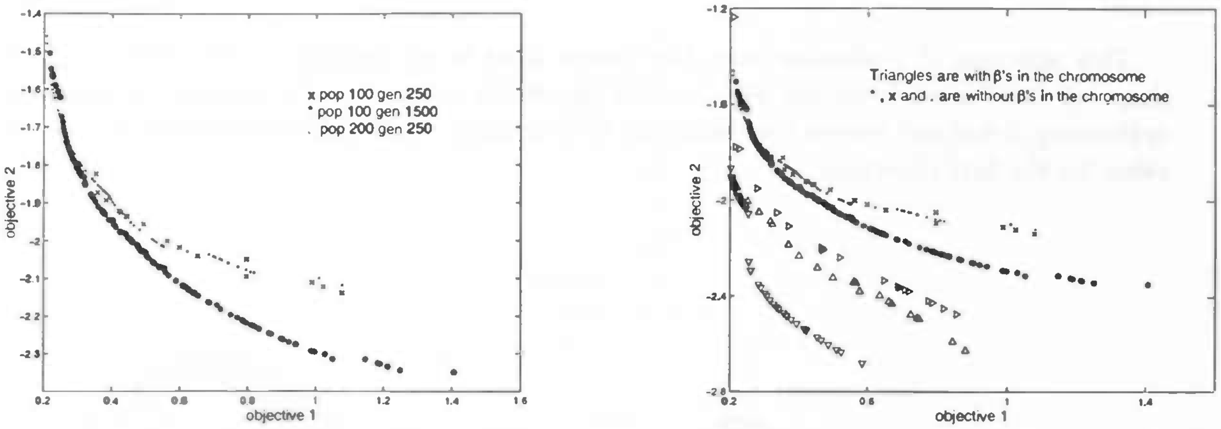
7.3.2 β's fixed at one and removed from the chromosome

Next the β's will be altogether left out of the chromosome. Now the chromosome consists of only 5 alleles. The non-dominated points found with three optimisations with this chromosome are shown in the left plot of figure 7.5.

Again there is not much difference between the fronts reached with population size 100 and 200. The front found after 1500 generations is much wider and lies a bit lower than the front found after 250 generations, both using 100 individuals.

As can be seen in the right plot in figure 7.5, this optimisation with the β's fixed at one in the chromosome results in a lower and therefore better Pareto front than the optimisation with the β's fixed at one but left out of the chromosome. Thus an optimisation with the β's included in the chromosome gives better results.

The settings for the recombination operators give an explanation for this result. With the β's fixed at one in the chromosome, about fifty percent of the mutation and crossover are likely to occur on the part of the chromosome that holds the β's. In that event the chromosome does



Three optimisations without β's in the chromosome.

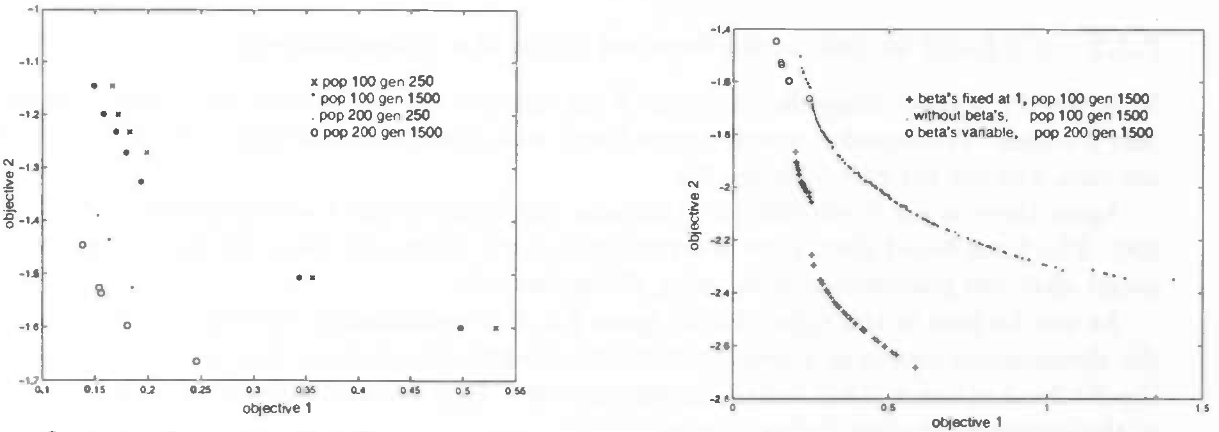
Comparing the results with and without the β's in the chromosome.

Figure 7.5: The results without β's in the chromosome and comparing the two different approaches.

not change because these alleles are fixed at one. Therefore the actual chance on mutation and crossover on the chromosome is reduced with the optimisation of the larger chromosome. This optimisation leads to better results then the optimisation with the shorter chromosome where the β's are left out of the chromosome. Therefore the chances on mutation and crossover apparently are set to high for the optimisation with the shorter chromosome. It can be concluded that the optimal values for the chances on mutation and crossover depends on the length of the chromosome.

7.3.3 β's variable

In the final optimisation the β's are put back into the chromosome and can vary between 0 and 1, just like $\alpha_1 ... \alpha_4$. The results of some optimisations using this chromosome are shown in the left plot of figure 7.6.



Optimisations with the β's varying between 0 and 1.

Comparing the different approaches.

Figure 7.6: The results with variable β's and all three approaches compared.

With this large chromosome the optimisation using 200 individuals in the population gives a better result then the optimisation using only 100 individuals. This can be explained by the fact that the gene pool is a lot bigger now that the β's can also vary between 0 and 1. Apparently a larger gene pool needs a larger population to give a good representation of all the possible genetic material. Thus an other optimisation is ran with 200 individuals in the population and 1500 generations in the evolution.

The results of the optimisations with the β's fixed at one or left out the chromosome give better results than the optimisation with the β's variable, as can be seen in the right plot in figure 7.6. In this figure the best fronts reached for all three optimisations are shown. The optimisation with the β's in the chromosome, but kept fixed at one gives the best results.

7.4 Conclusions

The optimisation of the parameters for the Runge-Kutta method in the flow solver Hexadap was a success. If some knowledge on possible good values is known before the optimisation, this can help the optimisation. In this case it was known that keeping the β's at one is a parameter setting used often with the Runge-Kutta method. The optimisation using this knowledge gave the best results. A shorter chromosome without these β's gave worse results, while in the calculations the β's were ascribed the value one. The settings for the recombination operators give an explanation for this result. Apparently, the chance on the events of mutation and crossover to happen are set too high for the optimisation with the shorter chromosome, as is explained in section 7.3.2. However, both optimisations with the β's fixed at one, with and without these β's in the chromosome, gave better results than the optimisation with variable β's in the chromosome. Therefore keeping the β's fixed at one, and therefore making use of that foreknowledge, always give a better result than letting the values for the β's vary.

Some great results were accomplished and these optimisations give reason to use a genetic algorithm more often with problems like this. Every method which makes use of some kind of parameter setting and for which some objectives can describe properties of an optimal working method, the parameter settings can be optimised. For example, the parameter settings of a genetic algorithm itself can be optimised using a genetic algorithm. That way the parameters as mentioned in chapter 6 and other parameters of the genetic algorithm can be optimised for a specific class of optimisation problems.

Chapter 8

Airfoil design optimisation

8.1 Introduction to the problem

Design optimisation of an airfoil is a multidisciplinary design optimisation problem if the different objectives come from different disciplines. In this chapter an airfoil design will be optimised using two objectives which determine the fitness of the airfoil in terms of aerodynamics. This is a single disciplinary multiobjective design optimisation problem. The difference between single- and multidisciplinary problems is that unlike objectives coming from several disciplines, the objectives coming from one discipline could possibly be reasonably spliced in one objective, turning the problem into a single objective problem. In section 8.6 suggestions are made how this single disciplinary multiobjective design optimisation can be used for solving a multidisciplinary optimisation.

The optimisation problem consists of the following objectives:

1. Maximise the lift coefficient C_l of the airfoil.

2. Minimise the drag coefficient C_d of the airfoil.

The constraints of this problem are:

1. The airfoil must have a non negative thickness.

2. The upper and lower half of the airfoil should meet at the trailing edge of the airfoil.

The first constraint ensures a physically correct shape of the airfoil without intersecting upper and lower half. The second constraint ensures a physical correct shape of the tail of the airfoil.

To calculate the necessary coefficients, i.e. the lift coefficient C_l and the drag coefficient C_d, the 2-D RANS flow solver Hitask, which is developed at NLR, is used. Because of the time necessary for the calculation with Hitask of these coefficients, the run is distributed over as many computers as there are individuals in the population. So with each generation each computer computes the coefficients for one individual. This way each generation takes as long as the longest computation time necessary for one individual from the population. Solving the Navier-Stokes equations on a coarse grid of 128 cells using 20 multi-grid cycles takes about 30 minutes. Some extra time is needed each generation to distribute the calculations over the computers, read in the results and let the current population evolve to the next generation, which together is less than 5 minutes.

8.2 Implementation

For these optimisations some changes to the implementation of SPEA had to be made. The chromosome, in this case, consists of ten parameters which define the airfoil shape through the parameterisation of the upper and lower half of the airfoil. This is called the GA-airfoil and will be used to create the final airfoil which the individual represents. The parameterisation is the following:

$$Z_{upper} = \sum_{n=1}^{6} a_n \cdot \mathbf{X}^{n-1/2} \text{ and } Z_{lower} = \sum_{n=1}^{6} b_n \cdot \mathbf{X}^{n-1/2}, \text{ with } \mathbf{X} \in [0,1] \tag{8.1}$$

This means that there are five free parameters for the upper half and five free parameters for the lower half. This sixth parameter for both halves is used to satisfy the second constraint. The ten parameters in the chromosome are the parameters which can be altered by mutation and/or crossover. In Matlab intervals for these parameters were determined to reduce the search space and ensure proper airfoils, with a thickness of about 0.1 unit.

$$a_1 \in [0.08, 0.12], a_2,\ a_3 \in [-0.12, 0], a_4,\ a_5 \in [-0.12, 0.12]$$
$$\text{and } b_1 \in [-0.12, 0.08], b_2,\ b_3 \in [0, 0.12], b_4,\ b_5 \in [-0.12, 0.12]$$

The geometry belonging to the parameters of an individual is calculated with Spea. Only airfoils that meet the first constraint and thus are considered feasible, are allowed. If either initialisation, mutation or crossover has supplied an infeasible design, this chromosome is replaced by a randomly generated new chromosome, until a chromosome which represents a feasible design is generated. The GA-airfoil, which is determined by the alleles and parametrisation (8.1) is used in two different ways to create the final airfoil that is represented by an individual. One way is to combine the GA-airfoil with the airfoil of the existing RAE2822 airfoil (RAE2822). This combination is called the Hitask-airfoil. This combination will look more or less, depending on which combination was chosen, like RAE2822. So this combination ensures a profile that is not too different from an existing, good, airfoil. The combination used is defined by

$$\alpha * Z_{RAE2822} + \beta * Z_{GA-airfoil} \;=\; Z_{Hitask-airfoil}. \tag{8.2}$$

In the right plot of figure 8.1 RAE2822 (-.-), an GA-airfoil (- -) generated with parametrisation (8.1) and the Hitask-airfoil (-), the combination of the first two airfoils using (8.2) with α set to 0.95 and β set to 0.1, are shown.

The parameter α determines the influence of RAE2822, β determines the influence of the GA-airfoil. Because the GA-airfoil can be both bigger and smaller than RAE2822, $\alpha + \beta$ must be larger than 1. This way, on the average, the Hitask-airfoil will be just as wide as RAE2822. α will always be chosen smaller than 1 because the GA-airfoil has a positive thickness, so the addition of the two airfoils would otherwise lead to a thicker Hitask-airfoil. With α set to a high value, for example 0.95, and thus the β set to a low value, for example 0.1, the resulting Hitask-airfoil will look a lot like RAE2822. With α set a bit lower on 0.7 and β set a bit higher to 0.4 the resulting Hitask-airfoil looks less like RAE2822 and the GA-airfoil will have more influence on the Hitask-airfoil.

The second way is to simply use the GA-airfoil as it is generated by FPspea as Hitask-airfoil.

With the airfoils determined in either way, the individuals are distributed over just as many workstations. On each workstation Hitask is started with as input the airfoil belonging to that individual.

A final check on the feasibility of an airfoil is given by Hitask. If Hitask for what ever reason can not generate a grid around an airfoil, C_l and C_d of that individual are set to a very low and high value, respectively.

A time-limit is set for all the Hitask calculations. If a Hitask calculation is not finished in time this could have several reasons. For example the machine on which the calculation is performed may have been slowed down by a nightly backup process. The coefficients of the individuals of which C_l and C_d are not calculated within the time-limit are set to the average value of the coefficients of that generation.

After the Hitask calculations have finished, Spea reads in C_l and C_d, the results from Hitask, which are kept close to the design variables responsible for these results. This way the objectives depending on these coefficients can be calculated everywhere in the algorithm between the calculation of the coefficients and the recombination of the individuals.

The mutation and crossover operators have been kept simple, i.e. one-point crossover and one-point mutation. The mutation probability Pm is set to 0.3 in all the optimisation runs and the crossover probability Pc is set to 0.8. The chromosome is not checked for its feasibility after mutation and crossover routines. The recombination operators do not often produce an infeasible design and checking this possibility would take up a lot of extra time.

If necessary it is possible to make a restart with an existing population. With a restart this population is read in FPspea after which the evolution can continue.

8.3 Euler versus Navier-Stokes

Hitask has the option to solve either the inviscid Euler equations to calculate C_l and C_d of an airfoil or the viscous Navier Stokes equations.

The Euler equations and Navier-Stokes equations give different solutions. The Euler equations do not acknowledge the existence of the boundary layer. A boundary layer makes an airfoil thicker and thus produces more drag. The Euler equations do not include the drag produced by shear stress on the surface of the airfoil, only the drag produced by air pressure is taken into account.

The boundary layer also influences the lift coefficient. Therefore, the Euler equations will return a lower drag coefficient than the Navier-Stokes equations. This effect is shown in figure 8.8.

With all optimisations in this chapter a quite coarse grid is used to speed up the calculations. Using this type of grid it will take Hitask between five and ten minutes to solve the Euler equations and to calculate C_l and C_d for an airfoil. Using the Navier-Stokes equations to do the same will take Hitask around thirty minutes.

In the first optimisations the Euler equations will be solved to calculate the C_l and C_d. Later on the more accurate and more expensive Navier-Stokes equations will be solved.

8.4 Results using the Euler equations

First the Euler equations are used to calculate the lift and drag coefficients. Three different kinds of airfoils were used as input for the Hitask calculations:

1. A combined airfoil using 8.2 with α set to 0.95 and β set to 0.4. This combination ensures smooth airfoils and therefore no troubles are to be expected with the grid generation from Hitask with these optimisations (section 8.4.1).

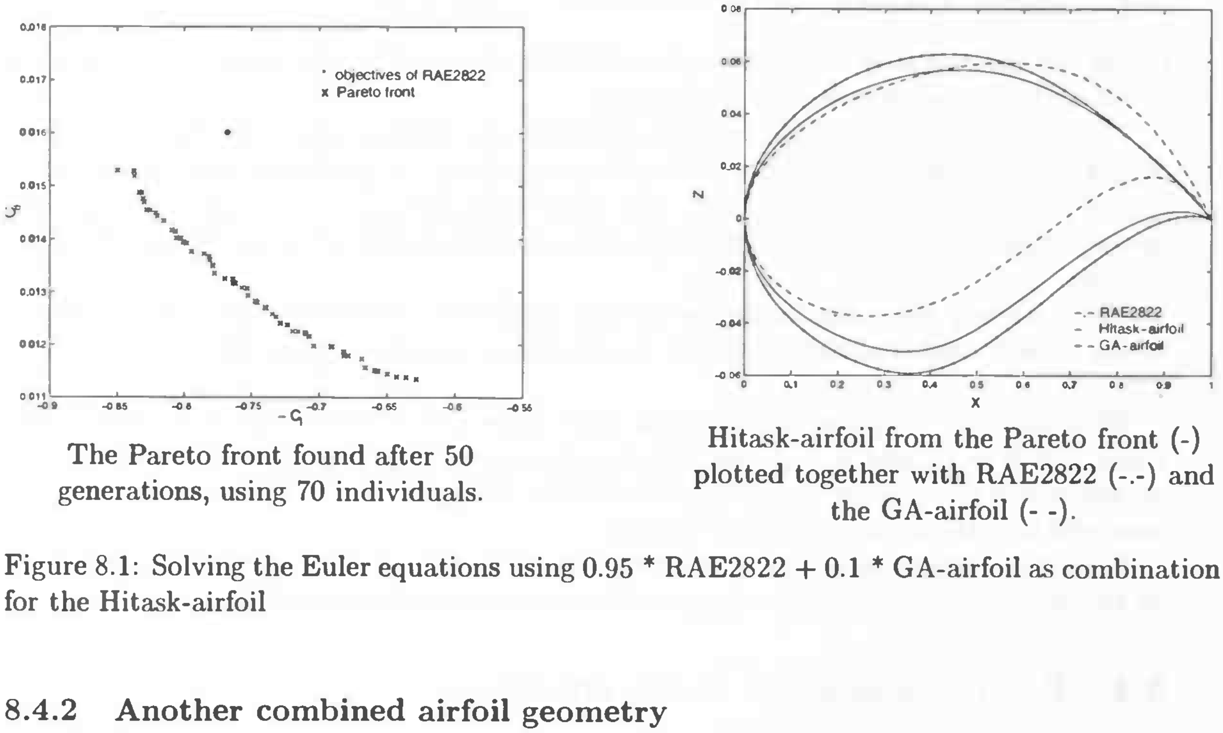
2. A combined airfoil using 8.2 with α set to 0.7 and β set to 0.4. This combination with quite a bit influence of the GA-airfoil may result in very realistic and good airfoils (section 8.4.2).

3. The GA-airfoil as it is calculated by the parametrisation (8.1). These airfoils have all the freedom within the intervals of (8.1), so with these airfoils the strength of the genetic algorithm can be fully exploited (section 8.4.3).

Apart from the parameters mentioned earlier in the chapter the parameter settings differ per optimisation and are specified in the text.

8.4.1 Combined airfoil geometry

With the first optimisations the combination of airfoils 8.2 was used with α set to 0.95 and β set to 0.1. The results of the optimisation using 70 individuals after 50 generations are shown in figure 8.1. A lot of non-dominated points are found which are well distributed over the Pareto front. The front lies a bit lower then the star which represents the fitness values of RAE2822, thus the non-dominated points achieve better results on the given objectives. In the right plot of figure 8.1, an airfoil represented by an individual in the Pareto front is shown. The used Hitask-airfoil has some nice properties also seen in other airfoils. This means that the genetic algorithm is capable of finding good and meaningful airfoils, which have good fitness values.



The Pareto front found after 50 generations, using 70 individuals.

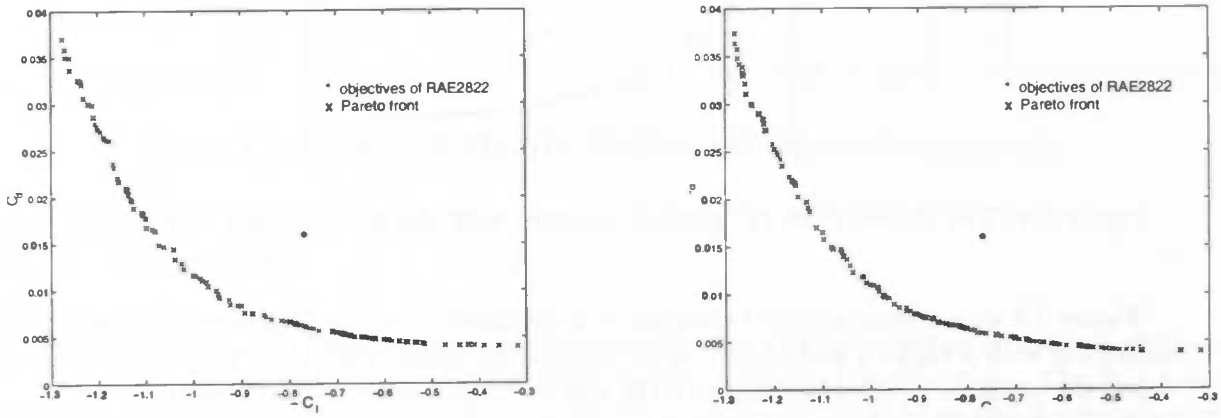
Hitask-airfoil from the Pareto front (-) plotted together with RAE2822 (-.-) and the GA-airfoil (- -).

Figure 8.1: Solving the Euler equations using 0.95 * RAE2822 + 0.1 * GA-airfoil as combination for the Hitask-airfoil

8.4.2 Another combined airfoil geometry

A combination with more influence of the GA-airfoil, but still using RAE2822 as a basis to form realistic airfoils gives the genetic algorithm enough freedom to evolve towards the Pareto front and gives the Hitask-airfoils enough properties of RAE2822 to ensure useful solutions. The combination 8.2 is used with α set to 0.7 and β set to 0.4. In figure 8.2 the front as it reached after 23 generations is plotted on the left side and the front as it is reached after 58 generations is plotted on the right side. It seems that this optimisation problem with influence of RAE2822 does not need more than 23 generations to get close to the Pareto front, because of the little difference in the position of these two fronts. When 23 generations are enough to get close to the *real* Pareto front, this optimisation can be run in reasonable time using the, much more expensive, Navier-Stokes equations, which is done in section 8.5. This section the Euler equations are used in Hitask.

This second combination gives a better result than the first combination, as can be seen in figure 8.7.

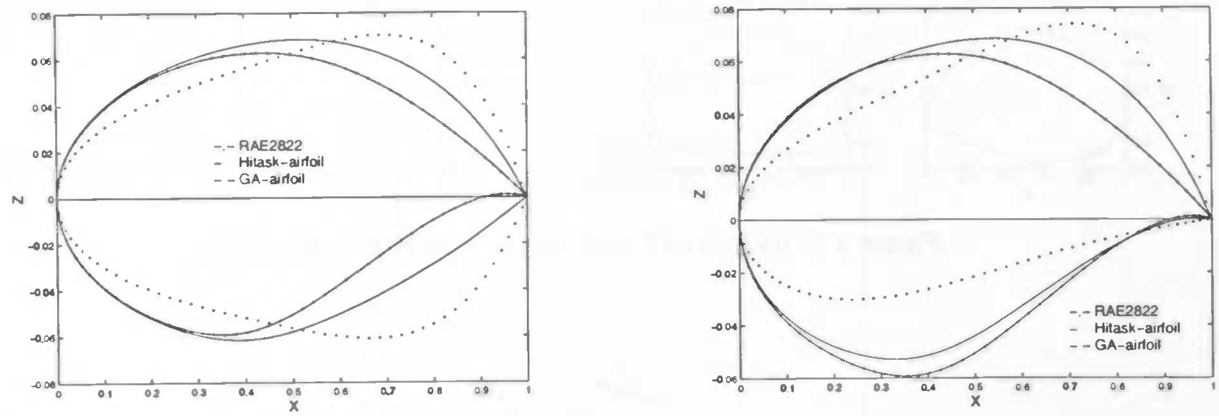


Pareto front found after 23 generations.

Pareto front found after 58 generations.

Figure 8.2: Optimisation using the second combination.

Figure 8.3 shows two airfoils from the Pareto front reached after 58 generations. The airfoil on the left has about the same C_l as RAE2822. The airfoil on the right has about the same C_d as RAE2822.



Airfoil with same C_l as RAE2822.

Airfoil with same C_d as RAE2822.

Figure 8.3: Two airfoils as result of an optimisation using the second combination.

8.4.3 The GA airfoil geometry

An optimisation using no combination but just the GA-airfoil as it is formed with parametrisation (8.1) gives a much wider Pareto front (see figure 8.4). The front also lies a bit lower than the Pareto fronts of both optimisations of the combined airfoils (see figure 8.7).

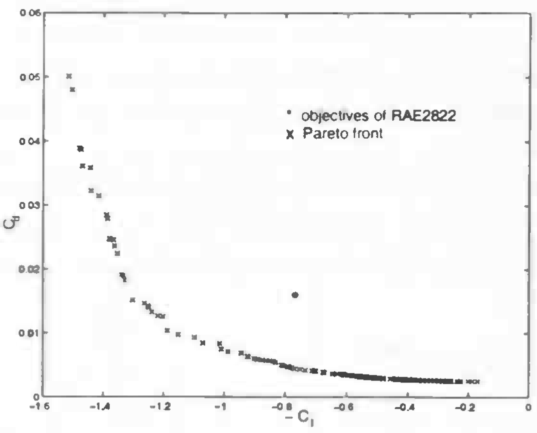


Figure 8.4: The Pareto front (x) plotted together with the fitness values of RAE2822 (*).

Figure 8.5 shows some airfoils belonging to individuals from the Pareto set. It starts with an individual with a high C_l and C_d and ends with an individual with a low C_l and C_d. Figure 8.6 shows the pressure profiles belonging to the airfoils from figure 8.5 respectively.

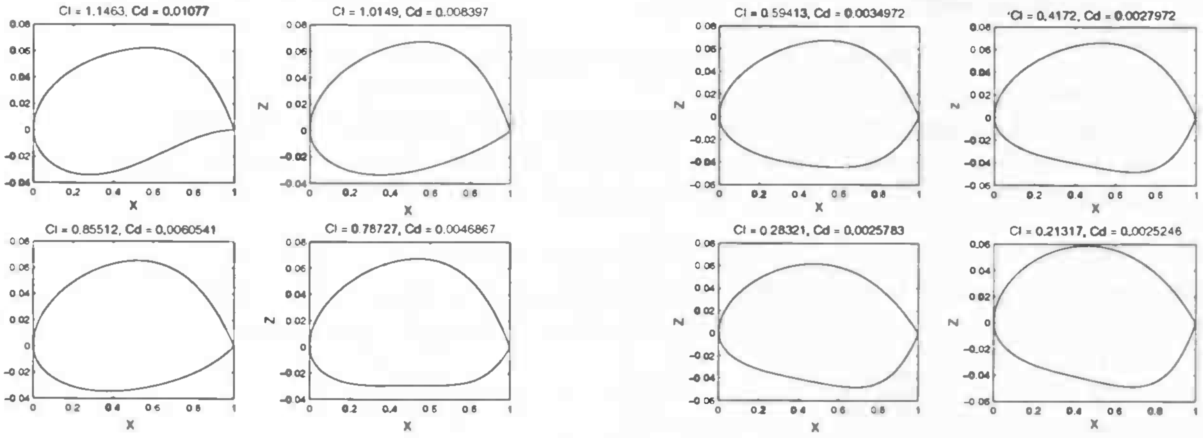


Figure 8.5: Airfoils of 8 individuals from the Pareto front.

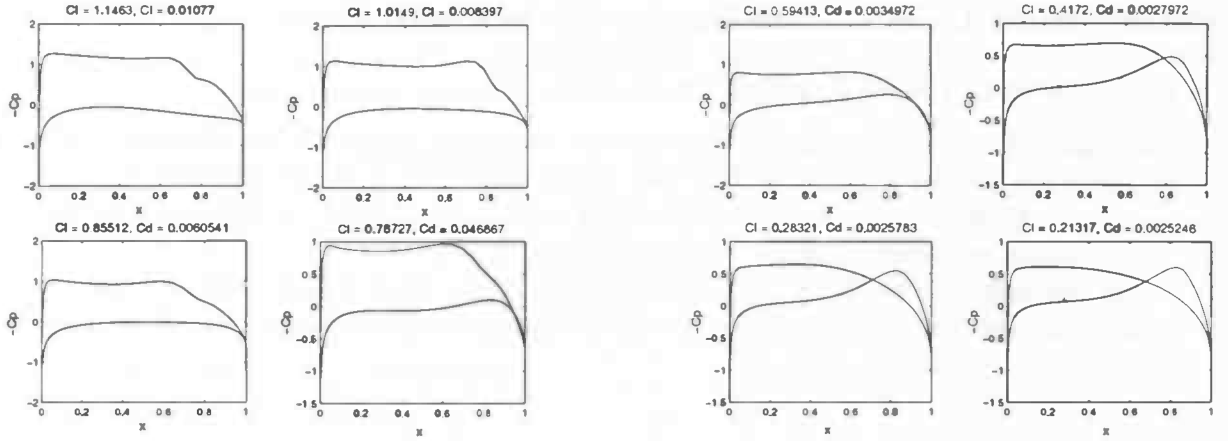


Figure 8.6: Pressure profiles for the airfoils from figure 8.5 respectively.

8.4.4 Conclusions

In figure 8.7 the Pareto fronts as they are reached using the different types of airfoils are plotted. The GA-airfoil without influences of RAE2822 results in the best Pareto front. The first combination, with high influence of RAE2822 gives the worst results and the second combination of airfoils results in a front in between. From these results can be concluded that with this optimisation problem, i.e. with these objectives, there is no reason to use a combination of RAE2822 and the GA-airfoil. With all optimisations using the Navier-Stokes equations the GA-airfoil will be used, without combining it with RAE2822. With a different optimisation with different or more objectives a combination described above may be useful. Such a combination can ensure that certain desired properties of the airfoil used to combine the GA-airfoil with are present in the population.

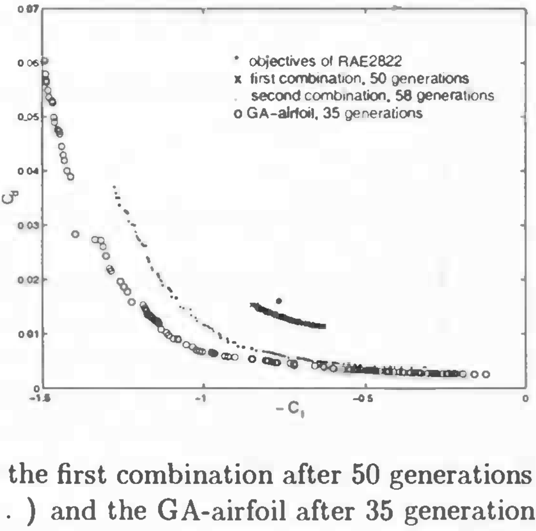


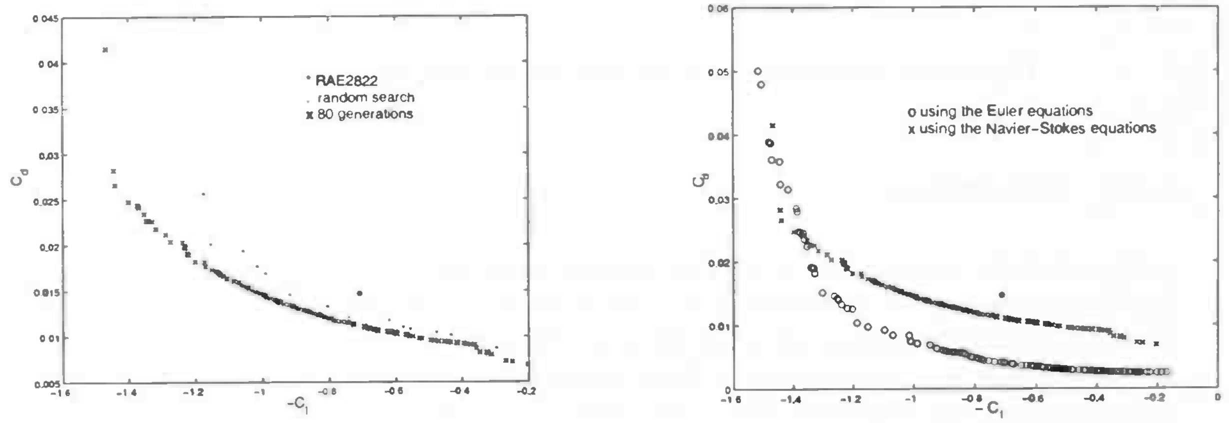
Figure 8.7: Pareto front of the first combination after 50 generations (x), the second combination after 58 generations (.) and the GA-airfoil after 35 generations (o).

8.5 Results using the Navier-Stokes equations

8.5.1 The GA airfoil geometry

The Pareto front found with the optimisation using the Navier-Stokes equations are shown in figure 8.8 On the left the front of this optimisation (x) is shown together with the front reached with random search (.). This front is created by plotting the initial population from this optimisation, which equals a random search with 70 tries.

On the right the resulting front of this optimisation (x) is shown together with the front found using the Euler equations (o). The differences in the position of the two Pareto fronts are explained in section 8.3.



Pareto front found after 80 generations (x) and using random search (.).

Pareto fronts found with Euler (o) and Navier-Stokes (x).

Figure 8.8: Some Pareto fronts found using the GA-airfoil.

Among the solutions there is a solution with $C_{l_{GA}} = 1.16 * C_{l_{RAE}}$ and $C_{d_{GA}} = 0.84 * C_{d_{RAE}}$, which is a significant improvement to fitness values of RAE2822. C_l and C_d from this airfoil are 0.8229 and 0.0123 respectively. The airfoil is shown in figure 8.9.

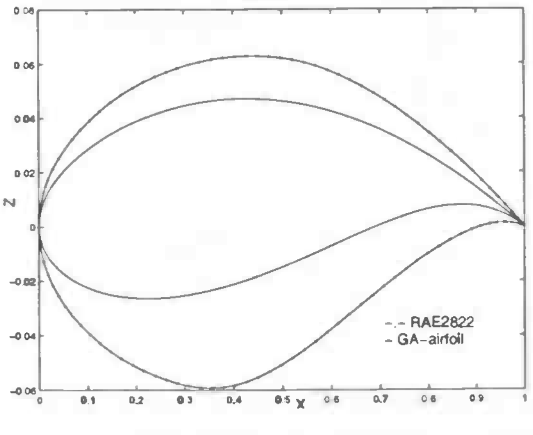


Figure 8.9: An airfoil from the Pareto set with better C_l and C_d then RAE2822.

Figure 8.10 shows some airfoils belonging to individuals from the Pareto set. It starts with an individual with a high C_l and C_d and ends with an individual with a low C_l and C_d. Figure 8.11 shows the pressure profiles belonging to the airfoils from figure 8.10 respectively.

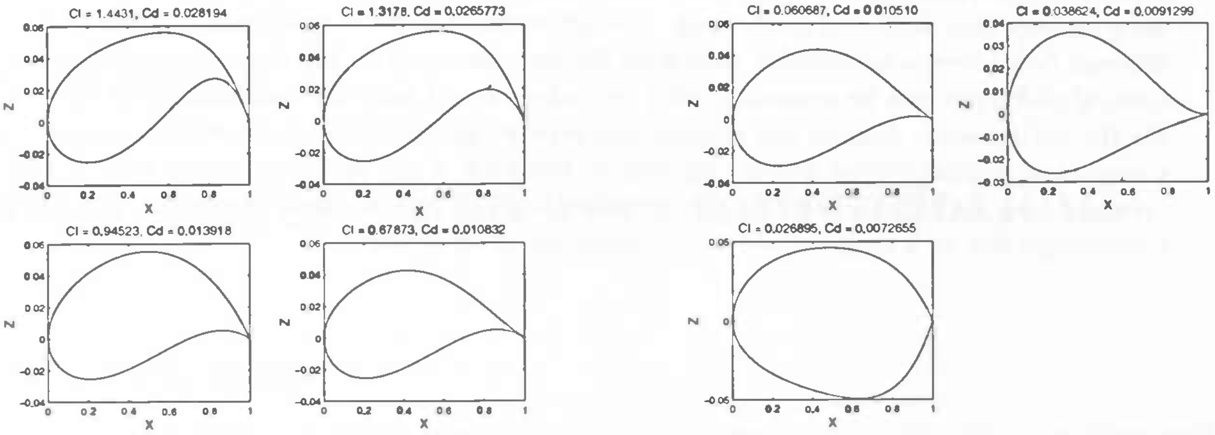


Figure 8.10: 7 airfoils from the Pareto set.

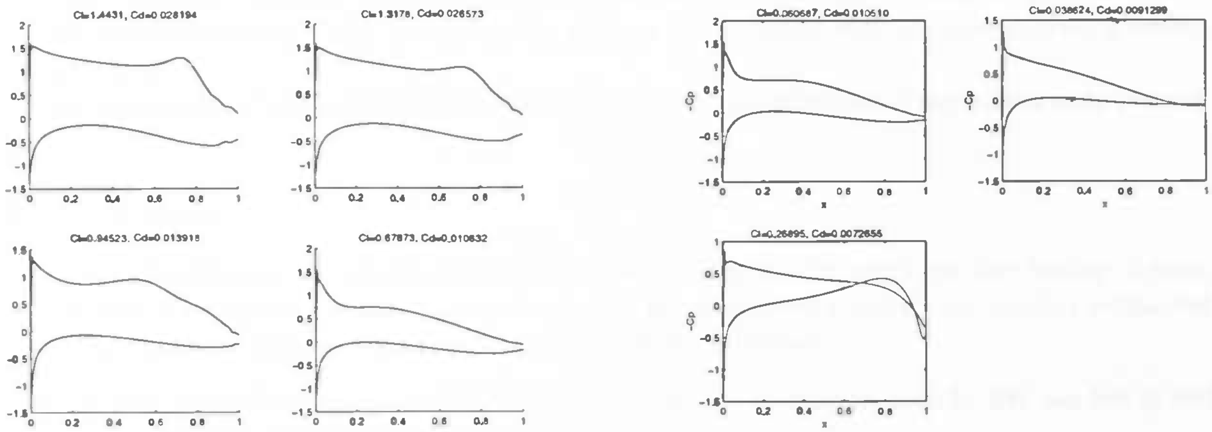


Figure 8.11: Pressure profiles of the airfoils shown in figure 8.10 respectively.

8.6 Multidisciplinary design optimisation

The single disciplinary multiobjective optimisation performed in this chapter can be extended to a multidisciplinary multiobjective design optimisation problem by adding one or more objectives which originate from a discipline different than aerodynamics. For example an objective from a different discipline is the volume of the airfoil, which is a measurement for the amount of fuel that can be stored in the airfoil. Another example is an objective on the strength of the airfoil.

Another way to involve other disciplines and/or choosing the final design from the Pareto front is to use them for the post-optimisation decision making. With the optimisation a Pareto front is found. From this front a single solution has to be selected as the best design. In making this decision, other objective(s) can be used to determine which of the non-dominated points of the Pareto set is the best. This objective again can come from a different discipline.

In a previous optimisation of an airfoil design at the NLR, a single objective genetic algorithm was used ([14]). The objective in that article consists of multiple objectives, including two objectives concerning the lift and drag coefficients of the airfoil. These objectives are spliced using appropriate weighting coefficients. The lift coefficient of the airfoil is kept fixed. Because a different flow solver was probably used with the optimisations in [14], the drag coefficients of the optimal airfoil can not be compared with the values found with the optimisation in this report. For the optimisation done in this chapter this article can therefore not be used as comparison of a single and multiobjective genetic algorithm. However, if the optimisations in this chapter are extended to a multidisciplinary design optimisation, [14] may be used to compare the results of a multiobjective to a single objective optimisation of an airfoil.

Chapter 9

Conclusions and recommendations

9.1 Conclusions

This report contains optimisations of a multiobjective design problems. This was done with FPspea, an extended version of SPEA which is a multiobjective genetic algorithm. GAs will not be the best algorithm to solve all optimisation problems, as is stated in the "No free lunch" theorem. However, because of its global search properties, its ability to find multiple solutions and its parallelisability, a GA will generally achieve good results with multiobjective optimisation problems.
Overall the results of the optimisations using FPspea as an optimisation algorithm were promising.

9.1.1 FPspea

No great difficulties were encountered with the use of FPspea. Although the fine tuning of some parameters of a GA can improve the results, a GA can give good results with roughly estimated values for those parameters. There are some points of attention:

- P_m is a powerful parameter. Set too high it will lead to random search. Set too low it will slow down the convergence rate considerably.
- The result of an optimisation can depend highly on the type of coding used. Several codings should therefore be tested.
- The result of an optimisation can be dependent of the seed used. An optimisation should therefore always be run several times with different seeds. This gives a clue on the distance between the *discrete* and the *real* Pareto front. The design problems in this report were optimised at least 5 times with different seeds.
- The elitism mechanism is an important mechanism that should always be present in a multiobjective GA.

9.1.2 Determining strategies for GAs

In chapter 6 an attempt was made to determine a strategy for GAs for specific type of optimisation problems. It can be concluded that it is possible to determine useful strategies for classes of optimisation problems. Several parameters were looked at of which some influence the results more than others. The optimal value of some parameters depend on the value of other parameters. More specific conclusions can be found at the end of chapter 6. Furthermore it is shown in chapter 7 that the optimal values of the chance on mutation and crossover depend on the length of the chromosome.

9.1.3 Design optimisation problems

In real problems the results of optimisations with FPspea were satisfactory. With the optimisation of the parameters of a Runge-Kutta method (chapter 7) it was shown that foreknowledge on the design can be useful. It was demonstrated that the optimal chance for mutation and crossover to occur depends on the length of the chromosome. Furthermore is was shown that FPspea can handle constraints well. The resulting design performs significantly better on the objectives of the optimisation problem and makes the Runge-Kutta method considerably faster.

With the optimisation of an airfoil (chapter 8) it was shown that FPspea is very well parallelisable. This property ensures that FPspea can very well be used for aerodynamic optimisation problems, were the fitness calculations with CFD-calculations can be very expensive. The resulting airfoil performs significantly better on the objectives than the RAE-2822 airfoil.

9.2 Recommendations

For the determinations of general strategies for GAs only the tip of the iceberg was dealt with in this report. A lot of work is still to be done on this subject. A general strategy should be determined for more test functions. More parameters should be taken into account. Special attention should be given to genetic drift, the convergence rate and the global aspects of the GA. The influences of the different recombination, mutation and ranking methods can also be examined. More specific recommendations are given at the end of chapter 6.

The optimisations on the design problems in chapter 7 can be extended to other optimisation problems involving parameter settings of a method or algorithm.

The optimisation of an airfoil design as performed in chapter 8 can be extended to multidisciplinary design optimisation as is stated at the end of that chapter.

Appendix A

Test functions

Before an algorithm can be tested first a good set of test functions should be determined. In this report the test functions formulated in [21] are used. These test functions are based on a paper written by Deb [4]. In this paper Deb describes the properties test functions should have and presents a formulation for a general two-objective optimisation problem. With this formulation problems having specific features can be created. The next section contains parts from this paper and gives some background to the test functions used in this report.

Two-objective test functions

Primarily, there are two tasks that a multiobjective GA should do well in solving multiobjective optimisation problems:

1. Guide the search towards the global Pareto-optimal region.
2. Maintain population diversity in the current non-dominated front.

The optimisation goal for a Multiobjective Optimisation Problem (MOP) may be reformulated in a more general fashion based on three objectives:

1. The distance of the resulting non-dominated front to the Pareto-optimal front should be minimised.
2. A good (in most cases uniform) distribution of the solutions found is desirable.
3. The spread of the obtained non-dominated front should be maximised, i.e., for each objective a wide range of values should be covered by the non-dominated solutions.

Convergence to the *real* Pareto-optimal front may not happen because of various reasons:

1. Multi-modality.
2. Deception.
3. Isolated optimum.
4. Collateral noise.

There is a difference between the difficulties caused by multi-modality and by deception. For deception to take place, it is necessary to have at least two optima in the search space (a true attractor and a deceptive attractor), but almost the entire search space favours the deceptive (non-global) optimum, whereas multi-modality may cause difficulty to a GA, merely because of the sheer number of different optima where a GA can get stuck to.

The following features of a multiobjective optimisation problem may cause multiobjective GAs to have difficulty in maintaining diverse Pareto-optimal solutions:

1. Convexity or non-convexity in the Pareto-optimal front,

2. Discontinuity in the Pareto-optimal front, and

3. Non-uniform distribution of solutions in the Pareto-optimal front.

Now we present a more generic two-objective optimisation problem which is constructed from single objective optimisation problems. Let us consider the following N-variable two-objective problem:

$$Minimize f_1(\vec{x}) = f_1(x_1, x_2, ..., x_m),$$

$$Minimize f_2(\vec{x}) = g(x_{m+1}, ..., x_N)h(f_1(x_1, ..., x_m), g(x_{m+1}, ..., x_N)).$$

The function f_1 is a function of m $(< N)$ variables $(\vec{x}_I = (x_1, ..., x_m))$ and the function f_2 is a function of all N variables. The function g is a function of $(N - M)$ variables $(\vec{x}_{II} = (x_{m+1}, ..., x_N))$, which do not appear in the function f_1. The function h is a function of f_1 and g function values directly. We avoid complications by choosing f_1 and g functions which only take positive values (or $f_1 > 0$ and $g > 0$) in the search space.

By choosing appropriate functions for f_1, g, and h, multiobjective problems having specific features can be created. This construction allows a controlled way to introduce such difficulties in test problems:

1. Convexity or discontinuity in the Pareto-optimal front can be affected by choosing an appropriate h function. This function tests an algorithm's ability to handle different types of the Pareto-optimal front.

2. Convergence to the true Pareto-optimal front can be affected by using a difficult (multi-modal, deceptive, isolated or others) g function, as already demonstrated in the previous section. This function tests an algorithm's ability to handle difficulties lateral to the Pareto-optimal front.

3. Diversity in the Pareto-optimal front can be affected by choosing an appropriate (non-linear or multidimensional) f_1 function. This function tests an algorithm's ability to handle difficulties along the Pareto-optimal front.

In this paper three tables with different types of function with examples and their effect are given for f_1, g and h. The Equations (Eqn) mentioned in the tables are listed after the tables.

Table 1: Effect of function f_1 on the test problem.

Function $f_1(x_1, ..., x_m)(> 0)$ Controls search space along the Pareto-optimal front			
	Type	Example	Effect
F1-I	Single-variable $(m = 1)$ and linear	$\delta_f + c_1 x_1$ $(\delta_f, c_1 > 0)$	Uniform representation of solutions in the Pareto-optimal front. Most of the Pareto-optimal region is likely to be found.
F1-II	Multi-variable $(m > 1)$ and linear	$\delta_f + \Sigma_{i=1}^{m} c_i x_i (\delta, c_i > 0)$	Non-uniform representation of Pareto-optimal front. Some Pareto-optimal regions are not likely to be found.
F1-III	Non-linear (any m)	Eqn A.8 for m = 1 or, $1 - e^{(-4r)} sin^4(5\pi r)$ where $r = \sqrt{\Sigma_{i=1}^{m} x_i^2}$	Same as above.
F1-IV	Multi-modal	Eqn A.1 with $g(x_2)$ replaced by $f_1(x_1)$ or other standard multi-modal test problems (such as Rastrigin's function, see Table 2)	Same as above. Solutions at global optimum of f_1 and corresponding function values are difficult to find.
F1-V	Deceptive	$f_1 = \Sigma_{i=1}^{m} f(l_i)$, where f is same as g defined in Eqn A.2	Same as above. Solutions at true optimum of f_1 are difficult to find.

Table 2: Effect of function g on the test problem.

<table>
<tr><th colspan="4">Function $g(x_{m+1}, ..., x_N)(> 0)$, say $n = N - m$
Controls search space lateral to the Pareto-optimal front</th></tr>
<tr><th></th><th>Type</th><th>Example</th><th>Effect</th></tr>
<tr><td>G-I</td><td>Uni-modal, single variable (n = 1), and linear</td><td>$\delta_g + c_2x_2$ $(\delta_g, c_2 > 0)$, or Eqn A.7 with $\gamma = 1$</td><td>No bias for any region in the search space.</td></tr>
<tr><td>G-II</td><td>Uni-modal and nonlinear</td><td>Eqn A.7 with $\gamma \neq 1$,</td><td>With $\gamma > 1$ bias towards the Pareto-optimal region and with $\gamma < 1$, bias against the Pareto-optimal region.</td></tr>
<tr><td>G-III</td><td>Multi-modal</td><td>Rastrigin: $1 + 10n +$
$\Sigma_{i=m+1}^{N}\{x_i^2 - 10cos(2\pi x_i)\}$
$x_i \in [-30, 30]$
Schwefel:
$1 + (6.5\pi)^2 n -$
$\Sigma_{i=m+1}^{N} x_i sin(\sqrt{|x_i|})$
$x_i \in [-512, 511]$
Griewangk:
$2 + \Sigma_{i=m+1}^{N} \frac{x_i^2}{4000}$
$-\Pi_{i=m+1}^{N} cos(\frac{x_i}{\sqrt{i}})$
$x_i \in [-512, 511]$</td><td>Many $(61^n - 1)$ local and global Pareto-optimal fronts
Many $(8^n - 1)$ local and one global Pareto-optimal fronts
Many $(163^n - 1)$ local and one global Pareto-optimal fronts</td></tr>
<tr><td>G-IV</td><td>Deceptive</td><td>Eqn A.2</td><td>Many $(2^n - 1)$ deceptive attractors and one global attractor</td></tr>
<tr><td>G-V</td><td>Multi-modal, deceptive</td><td>$g(u(l_i)) =$
$\begin{cases} 2 + e, & if\ e < l_i/2, \\ 1, & if\ e = l_i/2. \end{cases}$
where $e = |u(l_i) - l_i/2|$</td><td>Many $(\Pi_{i=m+1}^{N}\left[\begin{pmatrix} l_i \\ l_i/2 \end{pmatrix} + 2\right] - 2^n)$ deceptive attractors and 2^n global attractors</td></tr>
</table>

Table 3: Effect of function h on the test problem.

<table>
<tr><th colspan="4">Function $h(f_1, g)(> 0)$
Controls shape of the Pareto-optimal front</th></tr>
<tr><th></th><th>Type</th><th>Example</th><th>Effect</th></tr>
<tr><td>H-I</td><td>Monotonically nondecreasing in g and convex on f_1</td><td>Eqn A.3 or Eqn A.4 with $\alpha \leq 1$</td><td>Convex Pareto-optimal front</td></tr>
<tr><td>H-II</td><td>Monotonically nondecreasing in g and non-convex on f_1</td><td>Eqn A.4 with $\alpha > 1$</td><td>Non-convex Pareto-optimal front</td></tr>
<tr><td>H-III</td><td>Convexity in f_1 as a function of g</td><td>Eqn A.4 along with Eqn A.5</td><td>Mixed convex and non-convex shapes for local and global Pareto-optimal fronts</td></tr>
<tr><td>H-IV</td><td>Non-monotonic periodic in f_1</td><td>Eqn A.6</td><td>Discontinuous Pareto-optimal front</td></tr>
</table>

$$g(x_2) = 2.0 - exp\left\{-\left(\frac{x_2 - 0.2}{0.004}\right)^2\right\} - 0.8\ exp\left\{-\left(\frac{x_2 - 0.6}{0.4}\right)^2\right\} \tag{A.1}$$

$$g(u(l_i)) = \begin{cases} 2 + u(l_i), & \textit{if } u(l_i) < l_i, \\ 1, & \textit{if } u(l_i) = l_i \end{cases} \tag{A.2}$$

$$h(f_1, g) = \frac{1}{f_1} \tag{A.3}$$

$$h(f_1, g) = \begin{cases} 1 - \left(\frac{f_1}{\beta g}\right)^{\alpha}, & \textit{if } f_1 \leq \beta g \\ 0, & \textit{otherwise} \end{cases} \tag{A.4}$$

$$\alpha = 0.25 + 3.75\frac{g(x_2) - g^{**}}{g^{*} - g^{**}} \tag{A.5}$$

$$h(f_1, g) = 1 - \left(\frac{f_1}{g}\right)^{\alpha} - \frac{f_1}{g} sin(2\pi q f_1) \tag{A.6}$$

$$g(x_{m+1}, ..., x_N) = g_{min} + (g_{max} - g_{min})\left(\frac{\Sigma_{i=m+1}^{N} x_i - \Sigma_{i=m+1}^{N} x_i^{min}}{\Sigma_{i=m+1}^{N} x_i^{max} - \Sigma_{i=m+1}^{N} x_i^{min}}\right)^{\gamma} \tag{A.7}$$

The Pareto-optimal region occurs when g takes the value g_{min}. The parameter γ controls the biasness in the search space. If $\gamma < 1$, the density of solutions away from the Pareto-optimal front is larger.

$$f_1(x_1) = 1 - e^{-4x_1} sin^4(5\pi x_1),\ 0 \leq x \leq 1 \tag{A.8}$$

The above function has five minima for different values of x_1. The Pareto-optimal front is non-convex

The multiobjective function is defined over l bits, which is a concatenation of N substrings of variable size l_i such that $\Sigma_{i=1}^{N} l_i = l$, where $u(l_1)$ is the unitation of the first substring of length l_1. Unitation is the number of 1's in the substring. Note that minimum and maximum values of unitation of a substring of length l_i is zero and l_i, respectively.
The parameter q is the number of discontinuous regions in an unit interval of f_1.

The functions mentioned in the third column in each table are representative functions which will produce the desired effect mentioned in the respective fourth column. However, other functions of similar type (mentioned in the second column) can also be chosen in each case. While testing an algorithm for its ability to overcome a particular feature of a test problem, the complexity of the corresponding function (f_1, g, or h) should be varied and and the other two functions should be fixed at its easiest complexity level. For example, while testing an algorithm for its ability to find the global Pareto-optimal front in a multi-modal multiobjective problem, a multi-modal g function (G-III) and f_1 fixed as in F1-I and h as in H-I could be used. Similarly, using g function as G-I, h function as H-I, and by first choosing f_1 function as F1-I test a multiobjective optimisers capability to distribute solutions along the Pareto-optimal front. By only changing the f_1 function to F1-III (even with $m = 1$), the same optimiser can be tested for its ability to find distributed solutions in the Pareto-optimal front. This is because with a nonlinear function for f_1 function, there is a bias against finding some subregions in the

Pareto-optimal front. It will then be a test for a multiobjective optimisers ability to find those adverse regions in the Pareto-optimal front.

Some interesting combinations of these three functions will also produce significantly difficult test problems. For example, if a deceptive f_1 (F1-V) and a deceptive g function (G-IV) are used, it is likely that a multiobjective GA with a small population size will get attracted to deceptive attractors of both functions. In such a case, that GA will not find the global Pareto-optimal front. On the other hand, since not all function values of f_1 are likely to be found, some region in the Pareto-optimal front will be undiscovered. The G-V function for g has a massively multimodal landscape along with deception. This function introduces a number of different solutions having the same global optimal g function value. Corresponding to each of these globally optimal solutions for g function, there is one global Pareto-optimal front. In fact, in $f_1 - f_2$ space, all global Pareto-optimal fronts are the same, but solutions differ drastically. The sheer number of local Pareto-optimal fronts and deception in such a problem should cause enough difficulty to any multiobjective GA to converge to one of the global Pareto-optimal fronts. An interesting challenge in this problem would be to find all (or as many as possible) different globally optimal solutions for the g function.

Along with any such combination of three functionals, parameter interactions can be introduced to create even more difficult problems.

Appendix B

Program Description

B.1 Calling sequence

In the main file FPSpea.cpp the calling sequence of the methods is represented by

initialisation	Population	initRandom
		updateParetoSet
	HITASK	remoteROH
		readInput
evolution	spea	select
	Population	mate2
		mutate
	HITASK	remoteROH
		readInput
	Population	updateParetoSet

where the output methods are omitted and HITASK methods are only performed when CFD-calculations are necessary to determine the fitness of individuals. In this next figure FPspea without CFD-calculations is represented schematic:

Initialisation:

1 Initialisation of Population

2 Update of external set

- Non-dominated individuals are marked

The evolution, repeated until termination criterium is satisfied:

3 Selection

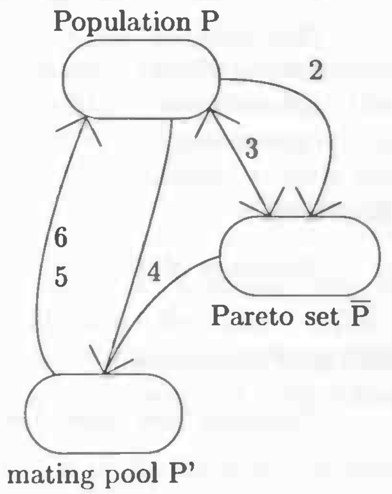
- Clustering

- Ranking

4 Recombination

5 Mutation

6 Update of external set



B.2 Classes

SPEA, Strength Pareto Evolutionary Algorithm, is an C++ algorithm. The structure of this algorithm exists of classes and subclasses. SPEA is built of the following classes:

SPEA

The class that takes care of selection (i.e. the filling of matingPool with parents) and assigns a fitness to all individuals in the method selection() with use of the method Population→updateParetoSet.

POPULATION

Container class that manages Individuals. Most of the functionality, like mating, clustering, etc., is already implemented such that subclasses only have to provide the actual container. In this class the Population control is handled. paretoSet is updated in updateparetoset(), individuals which show to few genetic diversity are filtered in avelinkCluster() and the mating is handled in mate2(). individuals is a property of the subclass VECPOPULATION, which is a vector of the object Individual.

INDIVIDUAL

Class that defines the interface of individuals used with multiobjective optimisation EAs. Once an a fitness value has been calculated, the value is stored in order to avoid further (possible expensive) calculations. Properties of this class are: the vector of booleans validObj, the vector of doubles objective, the double fitness, the enum Distance, the constant nrOfObjectives that is in an input variable and the adres of RandomNr randomNr. FPIND is a subclass of this class. In FPIND the actual methods are filled, like getObjective(), dominates() and covers(). These methods determine the fitness values of individuals and ranking method. The properties of this subclass are the constant nrOfAllele that is set to proper value in FPIND, the double maxDist and the object chromosome of the class FPCHROMOSOME.

CHROMOSOME

Base class that defines the interface of any kind of chromosome class. The propery of CHROMOSOME is the adres of RandomNr randomNr. FPCHROMOSOME is a subclass of this class for use with floating point problems. In this subclass the fpCrossover(), fpMutation() and other methods on chromosome level are defined. The properties of this subclass are the vector oplossing and the constant length, which is the length of oplossing.

RANDOMNR

Instances of this base class generate uniformly distributed pseudo-random numbers.

TIKEAFEXEPTION

Common base class of all exceptions that might be thrown from within the framework.

B.3 Structure

What was called the fitness values of an individual in the report, are the objectives of an individual in SPEA. What was called the rank of an individual in the report, is the fitness of an individual in SPEA. The objectives of the report are objective functions in SPEA.

Some objects are:

- *P* stands for `Population`, an object of the subclass VECPOPULATION,
- *PS* stands for `paretoSet`, an object of the subclass VECPOPULATION,
- `matingPool` is an object of the subclass VECPOPULATION,
- *I*, stands for `individual`, an object of the class INDIVIDUAL,
- *FPI* stands for `floatingPointIndividual`, an object of the class FPIND,
- *C* stands for `chromosome`, an object of the class CHROMOSOME and
- *FPC* stands for `floatingPointChromosome`, an object of the class FPCHROMOSOME.

B.3.1 Initialisation of Population

`Population::initRandom()`
The seed used for initialisation of the initial Population only depends on the number of individuals in that Population.
`length` is a property of *P*, which is the number of *I* in the vector `individuals`, a property of `VecPopulation`. `length` is an input variable (see section B.5.1). In `initRandom()` length *I*s are initialised.

> `Individual→FPInd→Chromosome→ FPChrom::fill()`
> The initialisation of *I* is done by the method `fill()` of *FPC*. This method fills the vector `oplossing`, which is a property of *FPC*, with random floating point values.

`SPEA::writeToFile()`
After initialisation, the *I* from the initial *P* are written to file. When an *I* is written to file, first the objectives of *I* are written down followed by the alleles of *I*.

B.3.2 Update of external set

`Population→updateParetoSet()::nondominatedPoints()`
The dominated *I* are deleted and a new *PS* is created with the non-dominated *I*.

> `Population→updateParetoSet()→nondominatedPoints()→FPInd::dominates()`
> In this method the boolean `bigger` is set to a value which determines whether the optimisation concerns a minimisation or a maximisation. `Individual::dominates()` is called, that calculates whether an individual is dominated by or dominates another individual. The method `Individual::getObjective()` is called.
>
> > `Individual→dominates()→getObjective()`
> > This method registers in `validObj`, a vector of booleans and a property of `Individual`, which objectives are already calculated. The method `FPInd→subGetObjective()` is called, that calculates the objectives of *I* and stores those values in `objective`, a vecor of doubles and a property of `Individual`.
> >
> > > `FPInd→subGetObjective()`
> > > The objectives of *FPI* are calculated. This is where the objective functions of the optimisation problem are stated.

Population→updateParetoSet()::deleteCovered()
The covered *I* are deleted from the *PS*.

Population→updateParetoSet()→ deleteCovered()→FPInd::covers()
In this method the boolean bigger is set to a values which determines whether the optimisation concerns a minimisation or a maximisation. Individual::covers() is called, to determine whether an individual is covered by or covers another individual.

Individual→dominates() →getObjective()
See earlier in this section.

B.3.3 Selection

SPEA→select()→Population::updateParetoSet()
See earlier.

SPEA→select()→paretoSet::aveLinkCluster()
The *PS* is reduced to the maximum size maxSize, which is an input variable (see section B.5.1), by deleting the *I* that are closest to each other in the objective space.

aveLinkCluster()→FPInd::distance()
With help of the double maxDist and the enum Distance, the type of distance to be calculated, the distance between two *I* is calculated. maxDist is a property of FPInd, Distance is a property of Individual. If the distance in the objective space is to be calculated, Individual::getObjective is called. (See earlier).

SPEA::select()
Determines the fitness of the individuals using Ind::covers(). The fitness is stored in the double fitness, that is a property of Individual. With the use of a binary tournament selection *I*s are selected as parents from the united set of the *P* and the *PS*, based on the fitness of the *I*. Fill matingPool with the selected *I*.

B.3.4 Recombination

matingPool::mate2()
Fills the new *P* with the children. Determines with randomNr.uniform0Max() two *I* from *PS* to become parents.

FPInd::mateWith()
Determines with randomNr.flipP() whether crossover will take place. In case of crossover FPChromosome::fpCrossover() is called.

FPChromosome::fpCrossover()
Here the double CrossoverPoint is determined with randomNr.uniform0Max() and two children, which are object of the class INDIVIDUAL, are created by performing some kind of crossover on the parents.

B.3.5 Mutation

Population::mutate()
For all length *I* FPInd::mutate() is called.

FPInd::mutate()
Determines whether mutation will take place with randomNr.flipP(). In case of mutation FPChromosome::fpMutation() is called on.

`FPChromosome::fpMutation()`
Here the double `mutationPoint` is determined with `randomNr.uniform0Max()` and for that allele a new value within appropriate range is determined.

After mutation the *I* from the current generation can be optionally added to file with `SPEA::writeToFile()`, see earlier this section.

B.3.6 Update of external set

See subsection B.3.2.

B.3.7 Output

After the evolution the *I*s from the `nondominatedSet`, an object of class VecPopulation, are written to file with `SPEA::writeToFile()`, see earlier this section.

B.4 Special routines

Special features build in to FPspea for the use of HITASK or just because they are handy are mentioned here:

- Restart
  It is possible to use the restart option. Instead of randomly initialising the initial `Population`, the final `Population` from a previous optimisation run is used as initial `Population`.

For HITASK:

- Dividing the CFD-calculations.
  `runonhosts` and `remoteROH` are two scripts to divide the CFD-calculations on several machines. `remoteROH` starts the `runonhosts` script on a remote workstation. `runonhosts` divides the CFD-calculations. Each individual is put in its own directory where the output files of HITASK are put. The output of HITASK relevant for the GA, the lift and drag coefficients, are put in the file `lift-drag.out`.

- Read in of lift- and drag coefficients.
  From the file `lift-drag.out` the coefficients are read in by `Population::readInput()`. The files `input.*` and `notsucc.*` make sure that the file `lift-drag.out` is complete before the data is read in.

B.5 Input files

B.5.1 Input files for FPspea

The making of the C++-files is managed by the file Makefile. This file also contains the input of FPspea. This input consists of 7 numbers:

- `nrOfAllele`: the number of variables in the chromosome that determine the design an individual represents;
- `nrOfObjectives`: the number of objectives that determine the optimisation problem;
- `nrOfGenerations`: the number of generations in the evolution;
- `popSize`: the number of individuals in the Population;
- `parSetSize`: the maximum number of individuals in the Pareto Set;
- `seed`: determining the values the random operators will produce and
- `tag`: a label used to distinguish the different optimisation runs.

B.5.2 Input files for HITASK

HTin.flow:

```
Standard NS
#----------
BluntTE    Adaption    Gridgen    MGcycl
   0          0           2         20
ICL     CLtar     Alpha       Re      Mach       Xtru       Xtrl     Tinf
 0        0.0       2.0    6.5e6      0.73       0.03       0.03     293
```

HTin.geo with 20 points defining the geometry:

```
# chromsome 00
   1.0000000       0.0000000E+00
   0.9283040      -3.1203817E-02
   0.7995878      -6.0426737E-02
   0.6840497      -6.9139503E-02
   0.5672981      -6.9465846E-02
   0.4497361      -6.5533056E-02
   0.3321484      -5.8949766E-02
   0.2162440      -4.9512387E-02
   8.5870251E-02 -3.2279204E-02
   1.8769074E-02 -1.5227628E-02
   0.0000000E+00  0.0000000E+00
   1.0045659E-02  1.1286175E-02
   6.9320090E-02  2.8989341E-02
   0.1710089      4.3436321E-02
   0.2858523      5.2223864E-02
   0.4266098      5.6043352E-02
   0.5448967      5.4033723E-02
   0.6861172      4.5474425E-02
   0.8247163      3.0249940E-02
   0.9295525      1.3783389E-02
   1.0000000      0.0000000E+00
```

B.6 Output files

B.6.1 Output files of FPspea

- `pop*`$_2$`_pop`
  This file contains the initial Population with $*_2$ individuals. After all, the random initialising of the Population uses the number of individuals in the Population ($*_1$) as seed.

- `objA*`$_1$`PopS*`$_2$`psS*`$_3$`.*`$_4$
  This file contains all the individuals of all generations of an optimisation run with $*_1$ alleles in a chromosome, maximal $*_3$ individuals in the paretoSet and tag $*_4$. It is optional to create this very large output file.

- `psA*`$_1$`popS*`$_2$`psS*`$_3$`.*`$_4$
  This file contains the individuals of the final `nondominatedSet`.

B.6.2 Output files of HITASK

- `HTout.table`
  This file contains the lift-, drag-, and moment coefficient (C_l, C_d and C_m) of the airfoil.

- `HTout.cp.table`
  This file contains the pressure profile belonging to the airfoil.

- `grid.dump`
  This file is created if HITASK cannot calculate the flow around an airfoil. This could be caused by properties of the geometry or by, for example, another process on a computer which slows down HITASK. If this file is created, the individual gets an average value for the coefficients C_l and C_d.

Bibliography

[1] Arakawa, Masao; Hagiwara, Ichiro; Nakayama, Hirotaka; Yamakawa, Hiroshi (1998), *Multiobjective Optimization Using Adaptive Range Genetic Algorithms with Data Envelopment Anlysis*, AIAA

[2] Atkinson Kendall E. (1989), *An Introduction to Numerical Analysis*, Wiley

[3] Crossley, William A.; Cook, Andrea M.; Fanjoy, David W. (1999), *Using the Two-Branch Tournament Genetic Algorithm for Multiobjective Design*, AIAA Journal, Vol. 37, no. 2

[4] Deb Kanpur, Kalyanmoy (1999), *Multi-Objective Genetic Algorithms: Problem Difficulties and Construction of Test Problems*, Evolutionary Computation, vol. 7, no. 3

[5] Doorly, D.J.; Peiró, J. (1998), *Distributed Evolutionary Computational Methods for Multiobjective and Multidisciplinary Optimization*, AIAA paper A98-39756

[6] Fonseca, Carlos M.; Fleming, Peter J. (1995), *An Overview of Evolutionary Algorithms in Multiobjective Optimization*, Evolutionary Computation, vol. 3, no. 1, p. 1-16

[7] Fonseca, Carlos M.; Fleming, Peter J. (1998), *Multiobjective optimization and Multiple Constraint Handling with Evolutionary Algorithms-Part I: A Unified Formulation*, IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, vol. 28, no. 1

[8] GAlib, `ftp://lancet.mit.edu/pub/ga`

[9] Gembicki, F.W. (1989), *Vector Optimization for Control with Performance and Parameter Sensitivity Indices.*, Ph.D. Dissertation, Case Western Reserve Univ., Cleveland, OH

[10] Goldberg, David E. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Publishing Company, Inc.

[11] Hüe, Xavier (1997), *Genetic algorithms for optimisation*, Edinburgh Parallel Computing Centre, The University of Edinburgh

[12] `intranet/~hexdap`

[13] De Kleermaeker, Simone (2000), *Multiobjective Design Optimisation and Genetic Algorithms: A Literature Survey*, NLR memorandum IW-2000-020, Amsterdam

[14] Kuiper, H.; van der Wees, A.J.; Hendriks, C.F.W.; Labrujère (1995), *Application of genetic algorithms to the design of airfoil pressure distributions*, NLR report TP95342L, Amsterdam

[15] Mäkinen, R.A.E.; Neittaanmäki, P.; Sefrioui, M.; Toivanen, J. (1997), *Parallel Genetic Solution for Multiobjective MDO*, Parallel CFD'96, pp. 352-459. Elsevier, Finland

[16] Miura, Hirokazu; Chargin, Mladen K. (1996), *A Flexible Formulation for Multi-objective Design Problems*, AIAA AIAA-96-4121-CP, p. 1187-1192

[17] Sobieszczanski-Sobieski, Jaroslaw; Haftka, raphael T. (1996), *Multidisciplinary Aerospace Design Optimization: Survey of Recent Developments*, AIAA paper no. 96-0330

[18] Surry, Patrick D.; Radcliffe, Nicholas J. (1997), *The COMOGA method: constrained optimisation by multi-objective genetic algorithms*, Control and Cybernetics, vol. 26, No.3, p. 391-412

[19] v/d Vegt, J.J.W.; van der Ven, H. (1998) *Discontinuous Galerkin finite element method with anisotropic local grid refinement for inviscid compressible flows*, J. Copmut Phys

[20] Vicini, Alessandro; Quagliarell, Domenico (1997), *Inverse and Direct Airfoil Design Using a Multiobjective genetic algorithm*, AIAA Journal, vol. 35, no. 9

[21] Zitzler, Eckart (1999), *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*, Ph.D. Dissertation, TIK, Zürich