OPTIMIZATION
An attempt at describing the State of the Art

Elke Pahl

International Center for Numerical Methods in Engineering (CIMNE)
Barcelona, Spain

2004

Abstract:
This paper is an attempt at describing the State of the Art of the vast field of continuous optimization. We will survey deterministic and stochastic methods as well as hybrid approaches in their application to single objective and multiobjective optimization. We study the parameters of optimization algorithms and possibilities for tuning them. Finally, we discuss several methods for using approximate models for computationally expensive problems.

Contents

List of Figures

Chapter 1

Introduction

1.1 Optimization Problems

The general optimization problem can be characterized as follows:

Given a decision space $\mathbf{X}$*, an objective space* $\mathbf{Y}$ *and a set of functions* $\mathbf{f} : \mathbf{X} \to \mathbf{Y}$ *(the objective functions) mapping each decision vector* $\mathbf{x}$ *to an objective vector* $\mathbf{y}$*, subject to the equality constraints* $g_i(\mathbf{x}) = 0$, $i = 1, \ldots, r$ *and the inequality constraints* $h_j(\mathbf{x}) \leq 0$, $j = 1, \ldots, s$*, find a parameter vector* $\mathbf{x}^* \in \mathbf{X}$ *that minimizes the components of the objective vector:*

$$\begin{cases} \min \mathbf{f}(\mathbf{x}) & \\ g_i(\mathbf{x}) = 0 & i = 1, \ldots, r \\ h_j(\mathbf{x}) \leq 0 & j = 1, \ldots, s \end{cases}$$

We can solve a maximization problem by solving the inverse minimization problem. Therefore, without loss of generality, we will assume minimization problems.

We distinguish between two types of optimization problems:

- **Combinatorial optimization:**
  Combinatorial optimization is a subset of discrete optimization, where $\mathbf{X}$ is discrete. In combinatorial optimization, $\mathbf{X}$ is usually finite and therefore there exist a finite number of solutions.
  Examples: Traveling Salesman Problem (TSP), Satisfiability Problem (SAT)

- **Continuous optimization:**
  In continuous optimization problems, we distinguish nonlinear and linear optimization. In linear optimization, $\mathbf{f}, g_i$ and h_j are linear. If one or more are nonlinear,

we face a nonlinear optimization problem.
Examples: Minimization/maximization of given functions,
optimization of material distribution, shape optimization

We can furthermore distinguish between single objective and multiobjective optimization problems. The former involve only one objective function, while the latter require the optimization of several objectives.

In this paper, we will focus on continuous optimization, investigating methods for solving both single objective and multiobjective problems.

The basic sequence of events when solving an optimization problem is to start from n initial points in the search space and to generate m new points in each iteration. Each point represents a possible solution of the problem. Every solution is evaluated and discarded or accepted as a new starting point. The procedure is continued until some criterion is fulfilled or the maximum number of iterations is reached.

In order to evaluate solutions, the user defines an *evaluation function*. In the case where the objective function is known explicitly, it can be used as the evaluation function. Otherwise, we have to specify an evaluation function reflecting the problem at hand as best as possible.

When solving optimization problems, we are faced with several difficulties:

- For most real-world problems, the search space is too large to apply exhaustive search. We can only test a small number of the possible solutions since we want to solve the problem in a given time frame.
- The objective function is usually highly nonlinear and can have several minima, so we have to take care not to get trapped in a local minimum that is not also a global minimum.
- In most practical optimization problems, the objective function is unknown. We can obtain single points of the objective function from experiments or simulations. Most probably, the chosen evaluation function does not accurately reflect the quality of the results. In this type of optimization problem, we are faced with a complete absence of a function relating the search parameters to the quality of the results.
- When using a model to approximate the problem (i.e. the exact objective function is unknown), the results might be distorted by noise. Noise can be systematic (running the same solution through the model more than once will yield the same (noise-impeded) results) or random (two runs of the same solution can yield different results or minute changes in a solution will yield very different results due to similator instabilities). Rounding errors adding up in the computer also produce noise.
- Running a simulation/experiment can be time-intensive, thereby further reducing the number of solutions we can afford to try.

- Some points in the search space are illegitimate. This is due to the points either violating at least one constraint (*infeasible points*) or causing the simulator to crash (*unevaluable points*).

1.2 Classification of Optimization Algorithms

Optimization algorithms can be classified using several different criteria. For example, we can differentiate between *deterministic* and *stochastic algorithms*. Deterministic algorithms will produce the same results if initiated at the same point (or points) of the search space. Stochastic algorithms incorporate randomness, so no two runs of the algorithms will produce the same result (at least not for a reasonable number of runs). Another classification criterion is the use of gradient information. Some algorithms require gradient information of the objective function during the search, others require only the value of the objective function. *Gradient methods*, if they converge, usually converge faster than *non-gradient methods*. Figure 1.1 classifies the most common algorithms into deterministic and stochastic methods, specifying whether gradient information is required by each method.

Of course it is also possible to combine deterministic and stochastic search methods into one algorithm. These algorithms are called *hybrid algorithms* and are currently enjoying a great popularity, since they allow the combination of the advantages of deterministic methods (fast convergence) with those of stochastic algorithms (broad exploration of the search space).

Another aspect distinguishing different algorithms is the number of solutions used as the basis for future exploration with each iteration. *Single solution algorithms* construct a single current best solution which is evaluated and on which the algorithm tries to improve in the next step. For example, all of the deterministic methods listed above are single solution algorithms. They require the serial evaluation of a function. Therefore, parallelizing these methods is not possible. *Multiple solution algorithms* work with several solutions at once. Most genetic algorithms are multiple solution algorithms. They are more suitable for parallelization, which enhances their otherwise rather low speed.

Of course it is possible to run several instances of a single solution algorithm at the same time, for example running a hill-climbing algorithm from several different starting points. This does not convert it into a multiple solution algorithm, however. The key to multiple solution algorithms is the fact that the solutions interact. This interaction can take place for instance in the construction of new solutions or while selecting the solutions to be preserved. We will go into further detail in Section 3.3.

In this paper, we will cover three important deterministic methods, as well as stochastic methods and hybrid methods. Emphasis is put on stochastic methods, since this field is far from fully explored and subject of much research in the optimization community. Furthermore, stochastic methods have several favorable properties qualifying them as a

- Deterministic methods:
  - Non-gradient methods:
    - Hill-climbing methods:
      evaluate point in neighborhood of current point, if new point evaluates better, it becomes the current point, if not, a different point is sampled (until better point is found)
    - Simplex Strategy:
      compute objective function for $n+1$ equidistant vertices, vertex with worst objective value is replaced by its reflection at the midpoint of the other n vertices.
    - Coordinate Strategy (Gauss-Seidel Strategy):
      uses coordinate axes as search directions, line search in each search direction
  - Gradient methods:
    - Conjugate Gradient (first derivative)
    - Quasi-Newton Methods (ex: Broyden-Fletcher-Goldfarb-Shanno) (first derivative)
    - Newton's Method (second derivative)
    - Levenberg-Marquardt (second derivative)
- Stochastic methods:
  - Random search
  - Genetic Algorithms (Non-gradient methods)
  - Non-genetic Algorithms (Non-gradient methods)

Figure 1.1: Classification of optimization algorithms

useful tool for difficult problems in optimization practice.

First, however, we will turn to an important theoretical issue regarding the superiority in performance of one algorithm over another.

1.3 No Free Lunch Theorems

Wolpert and *Macready* have proven that all optimization algorithms (including random search) perform equally well averaged over all possible problems. This statement is true for static problems as well as for time-dependent problems. In other words, for any algorithm, any elevated performance over one class of problems is exactly paid for in performance

over another class ("No Free Lunch" Theorems [28]).

This proof might seem disappointing at first, since it seems to imply that it is impossible to construct an optimization algorithm that is "better" than others found so far. This is however only partly true. Algorithm a can perform better than algorithm b on a specific problem, or even for a specific type of problem. Using a geometric interpretation of the NFL Theorems, Wolpert and Macready show that an algorithm's performance is determined by how "aligned" it is with the optimization problem at hand.

We will only give a short summary of the NFL theorems here. We will use the notation introduced by Wolpert and Macready:

$\mathcal{X}$:	search space
$\mathcal{Y}$:	space of possible cost values
$f : \mathcal{X} \to \mathcal{Y}$:	optimization problem (objective function)
$\mathcal{F} = \mathcal{Y}^{\mathcal{X}}$:	space of all possible optimization problems
$d_m = (d_m^x(1), d_m^y(1)), \ldots, (d_m^x(m), d_m^y(m))$:	set of m distinct visited points ("sample" of size m)
$\mathcal{D}_m = (\mathcal{X}x\mathcal{Y})^m$:	space of all samples of size m
$\mathcal{D} = \bigcup_{m \geq 0} \mathcal{D}_m$:	set of all possible samples
$a : d \in \mathcal{D} \to \{x \| x \notin d_{\mathcal{X}}\}$:	optimization algorithm

An optimization algorithm a is defined as a mapping from previously visited points to an unvisited point. This definition also holds for multiple solution algorithms which visit more than one new point during one iteration. The new points can be seen as being visited one after the other. It is assumed that an algorithm does not visit any point more than once. An algorithm visiting points more than once is wasting resources and can easily be improved by remembering which points have already been sampled. The NFL theorems hold for deterministic algorithms as well as stochastic algorithms.

In most practical cases, we do not know the exact objective function to be optimized. That is, we can generate an output $y \in \mathcal{Y}$ for any input $x \in \mathcal{X}$, but we do not know the exact landscape of the function to be minimized. The NFL theorems assume no knowledge of the objective function. Therefore, a probability distribution

$$P(f) = P(f(x_1), \ldots, f(x_{|X|}))$$

is introduced. This distribution $P(f)$, defined over $\mathcal{F}$, gives the probability that each $f \in \mathcal{F}$ is the actual optimization problem at hand. In considering $P(f)$ we acknowledge the fact that we do not know the exact objective function to be optimized.

The probability that a particular sample d_m is obtained using algorithm a iterated m times over the objective function f can now be expressed as $P(d_m^y | f, m, a)$. Theorem 1 of the NFL theorems states:

NFL Theorem 1: *For any pair of algorithms a_1 and a_2,*

$$\sum_f P(d_m^y|f, m, a_1) = \sum_f P(d_m^y|f, m, a_2)$$

In other words, when considering all possible objective functions, any two algorithms will on average perform equally. An algorithm performing better in one class of problems will necessarily perform worse in another class of problems. Since random search is also an algorithm according to the above definition, any algorithm performing better than random search for the problem it is tested on will perform worse than random search for some other problems.

This is an issue usually ignored by scientists inventing an optimization algorithm, testing it on a number of problems and subsequently stating that they have found a "better" algorithm based on the performance of their algorithm in the problems it was tested on. "It must be remembered that for any given problem it is always possible to produce a dedicated optimizer that will work better than any other, more general purpose approach: thus comparisons between methods should be treated with care." [11].

The probability of a particular algorithm a yielding the sought for sample d_m with a sample size m is

$$P(d_m^y|m, a) = \sum_f P(d_m^y|m, a)P(f)$$

Thus, for a non-uniform $P(f)$, it is possible to obtain better results for a certain problem with some algorithms than with others. The geometrical interpretation of the NFL theorems illustrates the connection between $P(d_m^y|m, a, f)$ and $P(f)$. The interested reader is refered to [28]. Unfortunately, in most practical cases the probability distribution $P(f)$ is impossible to obtain. We might have some information as to which type of function we are more likely to face, however determining the exact probability distribution is an infeasible task. Therefore, the NFL theorems, even though they are an essential piece of knowledge for everyone undergoing the task of designing optimization algorithms, are not of much practical use in terms of providing hands-on help to optimize the algorithm towards the task at hand.

1.4 Testing Optimization Algorithms

The "No Free Lunch" theorems have proven that across all problems, no search algorithm will perform better than pure random search. Investigations have however shown that it is possible to tune the parameters so that better than random results are achieved on a particular class of problem. This shows that it is possible to focus on a small subset of problems and find patterns that are effective in controlling the values of variation parameters.

In order to make a more informed decision about which algorithm to use and how to tune its parameters, it is advisable to first examine the problem at hand as closely as possible. In most cases we will not have the objective function at hand in closed form, but will depend on a simulation or experiments to give us results for solutions we propose. In this case, it is nevertheless possible to get a rough idea of the landscape of the objective function. A useful experiment to conduct is to use a simple algorithm and run it on the problem many times, plotting to which optimum the optimizer converges at every run. For example, we could use a deterministic algorithm started at many different points or a simple genetic algorithm run several times. In this way, we obtain a rough plot of the landscape of the evaluation function. We can see whether there are many local optima with a small basin of attraction each or whether we have a smoother landscape with fewer optima with larger basins of attraction. In the latter case, we might be able to tackle the problem with a greedier algorithm, while the former requires a more elaborate exploration of the search space.

Once the problem is sufficiently characterized, we choose the algorithm to be employed and test and tune it using test problems. These are problems for which the solution is already known. However, care must be taken that the test problems coincide sufficiently with the real-world problem at hand. Authors of algorithms often use a testbed previously used by others in order to be able to compare their results with those of other algorithms. This will however only assess how well the algorithm is suited for this particular class of problems. It is important to make sure the test problems used are as similar to the actual real-world optimization problem as possible.

We can judge the solutions an algorithm produces by how close they are to the known global optimum. In multiobjective optimization, this quality measure is not necessarily the optimal measure, as other aspects such as the diversity of the solutions are also important. We will expand on this issue in Section 3.4.

Besides the quality of the solutions an algorithm returns, we also have to take into account the time it needs to do this. Especially in applications where the objective function is not analytically known but evaluation of solutions is done via simulations (in the computer or in experiment), the number of objective function evaluations is usually the most time-consuming factor. In both single objective and multiobjective optimization, the number of objective function evaluations is therefore a critical measure.

Theorem 5 of the NFL theorems provides us with a performance measure of an optimization algorithm in conjunction with a specific problem. In short, it states that the probability of obtaining a "best-result" value larger than ϵ (assuming a minimization problem) is proportional to

$$\frac{1}{m+1}$$

in the limit of $|\mathcal{Y}| \to \infty$ and uniformly averaging over all cost functions. Thus, if the best-result-so-far of an algorithm drops slower than said distribution, it is most probably not well suited for the problem considered.

Chapter 2

Deterministic Methods

Deterministic methods have the advantage of usually converging quickly. However, they converge towards the closest local optimum, which in most cases will not be the global optimum. Therefore they need to be restarted several times, using different starting points, in order to find several optima. This weakens their computational advantage in comparison with stochastic methods. Whether or not it makes sense to use a deterministic method depends on the landscape of the objective function.

In this chapter, we will present two important deterministic methods: the *conjugate gradient method* (a gradient method) and *hill-climbing algorithms* (non-gradient methods).

2.1 Gradient Methods

2.1.1 Conjugate Gradient Method

A detailed and comprehensible outline of the Conjugate Gradient Method can be found in [23]. Here we will summarize the algorithm shortly, since it is one of the most successful deterministic gradient-based techniques.

The *Linear Conjugate Gradient Method* is used to minimize quadratic forms, i.e. functions of the form

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} + c$$

where $\mathbf{A}$ is a matrix, $\mathbf{x}$ and $\mathbf{b}$ are vectors and c is a scalar constant. If $\mathbf{A}$ is symmetric and positive-definite, $f(\mathbf{x})$ is minimized by the solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$. During the iteration, a new point is found from the previous one using the rule

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \, \mathbf{d}_i$$

where $\mathbf{d}_i$ are the *search directions*. The idea is to choose them such that they are A-orthogonal, that is conjugate with respect to A. In the linear case, this results in taking only one step in every direction. Thereby the optimum can be found in n steps for an n-dimensional linear problem.

The *Nonlinear Conjugate Gradient Method* can be used to minimize any continuous function $f(\mathbf{x})$ for which the gradient $f'(\mathbf{x})$ can be computed.

The search directions $\mathbf{d}_i$ can be found by Gram-Schmidt conjugation of n linearly independent vectors $\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{n-1}$.

$$\mathbf{d}_i = \mathbf{u}_i + \sum_{k=0}^{i-1} \beta_{ik}\, \mathbf{d}_k$$

where the β_{ik} are defined as

$$\beta_{ik} = -\frac{\mathbf{u}_i^T \mathbf{A}\, \mathbf{d}_k}{\mathbf{d}_k^T \mathbf{A}\, \mathbf{d}_k}$$

This algorithm as it stands is computationally expensive, since all search vectors must be kept in memory to construct the next one and $O(n^3)$ operations are required.

The solution is to use *Krylov subspaces* as search spaces. This is achieved by constructing the search directions by Gram-Schmidt conjugation of the residuals, defined by

$$\mathbf{r}_i = -f'(\mathbf{x}_i)$$

that is to set $\mathbf{u}_i = \mathbf{r}_i$. Thereby, most of the β_{ij} disappear and the complexity per iteration is reduced from $O(n^2)$ to $O(m)$, where m denotes the number of nonzero entries of A. The β_{ij} are zero for all $i > j+1$, therefore they can be written as $\beta_i = \beta_{i,i-1}$. The algorithm is outlined in Figure 2.1.

Finding α_i that minimizes $f(x_i + \alpha_i\, d_i)$ is done via any algorithm that computes the zeros of the expression $[f'(x_i + \alpha_i d_i)]^T d_i$.

The two different methods of obtaining β_i correspond to the *Fletcher-Reeves* formula and the *Polak-Ribière* formula, respectively. The latter often converges more quickly, but it is not guaranteed to converge. It has to be "restarted" if $\beta_i < 0$. Therefore, $\beta_i = \max\{\beta_i, 0\}$.

Since the Conjugent Gradient Method generates n conjugate vectors in an n-dimensional space, it should be restarted every n iterations. Furthermore, *preconditioning* can be used to enhance the performance of the algorithm. The interested reader is referred to [23].

Outline of the algorithm:

begin
Choose start vector x_0
$d_0 = r_0 = -f'(x_0)$
repeat
Find α_i *that minimizes* $f(x_i + \alpha_i\, d_i)$
$x_{i+1} = x_i + \alpha_i d_i$
$r_{i+1} = -f'(x_{i+1})$
$\beta_{i+1} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$ *or* $\beta_{i+1} = \max \left\{ \begin{array}{c} \frac{r_{i+1}^T (r_{i+1} - r_i)}{r_i^T r_i} \\ 0 \end{array} \right.$
$d_{i+1} = r_{i+1} + \beta_{i+1} d_i$
until (*termination-condition*)
end

Figure 2.1: Outline of the Nonlinear Conjugate Gradient Method

2.2 Non-gradient Methods

2.2.1 Hill-climbing Algorithms

2.2.1.1 Classical hill-climber

The classical hill-climbing algorithm is a single solution algorithm. During each iteration, a new point in the neighborhood of the current point is sampled. If this new point achieves a better fitness than the current point, it replaces the current point. If not, a different point is sampled. This procedure is continued until either a minumum (local or global) has been found or until the user-defined maximum number of iterations have been exhausted.

Different hill-climbing algorithms differ mainly in their choice of the next point to be evaluated. The *steepest ascent hill-climbing* algorithm for example samples all points in the neighborhood of the current point and then chooses the one with the highest fitness to compete with the current point. If improvement takes place, the old point is replaced with the new point, if not, the current point already represents a local optimum and the algorithm terminates.

An alternative is to sample one point after the other, until a better point than the current one is found. This is the procedure outlined in the above pseudo-code. Note that the new point might not present the best improvement possible, it simply guarantees any improvement at all.

Outline of the algorithm:

begin
pick starting point P
evaluate fitness of P
repeat
select new point P′ from neighborhood of P
evaluate fitness of P′
if *fitness(P′) > fitness(P)*
then $P \leftarrow P'$
else
continue
until (*termination-condition*)
end

Figure 2.2: Outline of a classical hill-climbing algorithm

2.2.1.2 Iterated hill-climber

Iterated hill-climbers start from several different points in order to "escape" from local optima. The algorithm finds several local optima and remembers the best one found so far. At the end of the run, this is is the the closest approximation of the global optimum that we have. Although the algorithm uses several starting points during the course of a run, it is also a single solution algorithm since it works on only one point at a time. When the possibilities of this point are exhausted, it moves on to the next point. The outline of the algorithm is shown in Figure 2.3.

2.2.2 Downhill Simplex Method

The *Downhill Simplex Method* was invented by *Nelder* and *Mead* in 1965. A *simplex* is a geometrical figure, which, in an n-dimensional space, consists of $n+1$ points. It is thus an $n+1$-sided polygon. The method works on *reflection*, *expansion* and *contraction* of the simplex. It is a multi solution algorithm. For a detailed explanation and some graphical representations of the procedure, see [10].

The simplex is constructed using the solution vectors. We need $n+1$ solutions to construct the simplex. Subsequently, every solution is evaluated. The point corresponding to the worst solution $\mathbf{x}_{\text{worst}}$ is then reflected through the opposite face of the simplex to generate a trial point $\mathbf{x}'$, maintaining the volume of the simplex:

$$\mathbf{x}' = \frac{2}{n} \sum_{i=1}^{n+1} \mathbf{x}_i - \left(\frac{2}{n} + 1 \right) \mathbf{x}_{\text{worst}}$$

The fitness of the trial point $f(\mathbf{x}')$ is evaluated. There are four possible outcomes:

Outline of the algorithm:

```
begin
    initialize best
    repeat
        local ← FALSE
        pick starting point v_c at random
        evaluate v_c
        repeat
            select new point v_n from neighborhood of v_c
            evaluate fitness of v_n
            if fitness(v_n) > fitness(v_c)
              then v_c ← v_n
            else local ← TRUE
        until local
        if v_c is better than best
        then best ← v_c
    until (termination-condition)
end
```

Figure 2.3: Outline of an iterated hill-climbing algorithm

1) $f(\mathbf{x}_{\text{2nd\_worst}}) < f(\mathbf{x}') < f(\mathbf{x}_{\text{best}})$:
Should the point $\mathbf{x}'$ yield a fitness value that lies between the fitness of the worst and the second best solution, the worst point is replaced by the new point.

2) $f(\mathbf{x}') \geq f(\mathbf{x}_{\text{best}})$:
Should the new point evaluate better or equal to the best solution, the simplex is extended in the direction of $\mathbf{x}'$, to see if a further improvement can be achieved. The hereby obtained solution $\mathbf{x}''$ is evaluated. The better of the two solutions $\mathbf{x}'$ and $\mathbf{x}''$ replaces the worst solutions. In the case of $\mathbf{x}''$, the volume of the simplex expands.

3) $f(\mathbf{x}') \leq f(\mathbf{x}_{\text{worst}})$:
Construct a point $\mathbf{x}''$ lying between the worst solution and the average of the other simplex points, exluding the worst solution. If $f(\mathbf{x}'') > f(\mathbf{x}')$, $\mathbf{x}''$ replaces the worst solution. If not, the simplex is contracted in all directions around the best solution.

4) $f(\mathbf{x}_{\text{worst}}) < f(\mathbf{x}') \leq f(\mathbf{x}_{\text{2nd\_worst}})$:
Construct a point $\mathbf{x}''$ lying between $\mathbf{x}'$ and the average of the other simplex points, exluding the worst solution. If $f(\mathbf{x}'') > f(\mathbf{x}_{\text{worst}})$, $\mathbf{x}''$ replaces the worst solution. If not, the simplex is contracted in all directions around the best solution.

This process of reflection, extension and contraction is continued until either an optimum is encountered or the predefined number of iterations is reached.

Chapter 3

Stochastic Optimization

3.1 Classification and Terminology

Stochastic optimization aims at minimizing/maximizing a function, incorporating randomness into the search procedure. Therefore, two runs of a stochastic algorithm will, in general, not yield the same result. The use of randomness allows the algorithms to escape local optima and to perform a broader search in the search space than deterministic methods. They are also better at handling noise (see Chapter 3.3.5).

Interest in stochastic methods has increased rapidly in the last decades. Many methods have been developed and ideas have been borrowed, exchanged and modified across all of these approaches. As a consequence, it has become difficult to classify the different methods unambiguously into categories. Categories often encountered in the literature are

- Evolutionary Algorithms (EA)
- Genetic Algorithms (GA)
- Evolutionary Programming (EP)
- Evolutionary Strategies (ES)
- Simulated Annealing (SA)
- Genetic Programming (GP)

Definitions vary significantly across the literature. For example, we find in [9]: *Evolutionary algorithm* is an umbrella term used to describe computer-based problem solving algorithms which use computational models of some of the known mechanisms of evolution as key elements in their design and implementation.

This definition is not very helpful, as it applies to all stochastic methods, if we consider all "mechanisms of evolution", these being

- recombination
- mutation
- selection

We will group stochastic methods into two groups: *Genetic Algorithms* and *Non-genetic Algorithms*. Genetic algorithms model evolution at the level of individuals of a species. Individuals recombine amongst each other, generating offspring. The offspring might also undergo mutation. Non-genetic algorithms model evolution at the level of species. Species do not recombine amongst each other. Each species might undergo change (mutation), but no information is shared between species. We feel this differentiation is adequate and justified, as an algorithm obtaining new solutions by merely varying prior solutions does not capture the essence of genetic evolution, which is the exchange of information between individuals via recombination.

However, since the goal of this paper is to provide a systematic overview of stochastic optimization methods and not to manifest new definitions, we will proceed as follows: The first part of this chapter discusses evolutionary operators and parameters and explores the possibilities they provide. We will then turn to the tuning of parameters, diversity preservation in the search space and to archiving strategies. Finally, we will introduce some specific algorithms, pointing out their use of the different operators and parameters discussed.

3.2 Ergodicity in Search

Ergodicity is the property characterizing a system that tends in probability to a limiting form that is independent of the initial conditions. Thus, independent of which point the algorithm starts from, it will give the same end result as time tends towards infinity.

Monte Carlo Markov Chains (MCMC) are ergodic. They sample points from the search space, following proposal mechanisms and acceptance rules. The frequency with which a point is visited is proportional to its probability in the search space. If run for an infinitely long time, a MCMC will give the probability distribution of all points in the search space.

The transition function of the Markov Chain must be ergodic. It is defined by the proposal mechanism together with the acceptance rule. In discrete optimization, the transition function is a matrix. In order for it to be ergodic, it has to posess one eigenvector with the associated eigenvalue of one and its remaining eigenvectors have to have an eigenvalue of zero. Then the vector of probabilities of all possible solutions Π is a fixed point of the transition function. Furthermore, the transition function has to fulfill the property of

detailed balance. This means that at a given time t it must be equally probable to jump from point a to point b as from b to a.

Proposal mechanisms propose the next point to be visited by the algorithm. An acceptance rule governs whether a proposed point is accepted as the next point. MCMC uses the Metropolis acceptance rule

$$P(X_{t+1} = b) = \begin{cases} 1, & p(b) > p(a) \\ \frac{p(b)}{p(a)}, & \text{otherwise} \end{cases}$$

going from point a to point b, where $P(X_{t+1})$ is the probability that point b will be accepted and $p(a)$ and $p(b)$ are the probabilities of points a and b, respectively. For a symmetric proposal distribution, the Metropolis acceptance rule guarantees that Π is a fixed point of the transition matrix. Should the proposal distribution not be symmetric, this has to be accounted for in the acceptance algorithm for the transition function to remain ergodic.

When using MCMC for optimization, the probability distribution is replaced by the objective function of the solutions, which has to be integrable. Points are visited with a probability proportional to how they are evaluated by the objective function. MCMC has to the best knowledge of the author not yet been applied to multiobjective optimization, as the presence of several objective functions requires different acceptance rules and the theoretical background for ergodicity in a multiobjective environment has not been investigated so far.

The advantage of MCMC is that it is guaranteed to visit every point in the search space in the limit and will therefore not get stuck in local optima. Its main disadvantage is that it is very slow. A smooth fitness function with few local minima can be optimized far more quickly with a greedier algorithm, heading more quickly for points with a higher fitness. It is important to note though that an algorithm with a higher greediness than the MCMC will compulsorily be less likely to evaluate points with low fitness. We are faced with a trade-off between convergence speed and exploration of the search space. Therefore, the landscape of the fitness function must always be kept in mind when designing the search algorithm to be used.

3.3 Basic Scheme of Genetic Algorithms

Most stochastic methods are *multiple solution algorithms*. They work on a *population* of solutions, called *individuals*, that strive for survival and for *reproduction*. The basic unit of evolution is the individual. Time is divided into discrete steps, called *generations*. At each generation some new individuals are generated. Each individual in the environment is evaluated by assigning to it a measure of its *fitness* in the environment. The goal of every algorithm is to find an individual of maximal fitness.

The basic scheme of a genetic algorithm can be outlined as follows [9]:

Outline of the algorithm

begin
initpopulation P(t)
evaluate P(t)
repeat
P':=*selectparents* P(t)
recombine P'(t)
mutate P'(t)
evaluate P'(t)
P:= *select* (P, P'(t))
until (*termination-condition*)
end

Figure 3.1: Basic scheme of a genetic algorithm

Non-genetic algorithms do not include the step *recombine* P'(t). Individual algorithms differ in the implementation of the operators and parameters, which we will discuss in the next section.

3.3.1 Operators and Parameters

Stochastic algorithms are controlled by a set of ***parameters***. These are

- representation
- evaluation function
- population (size, topology, initialization)
- halting criterion
- constraint handling

Furthermore, all algorithms apply a set of ***operators***. These are

- generating offspring
  - mutation
  - inversion
  - recombination
- selection
  - mating selection
  - environmental selection

Keeping the NFL Theorems in mind, we know that an optimum set of parameter choices for all problems does not exist. The parameters have to be tuned to the specific problem at hand. In the same way, parameter values tuned to a specific problem will not work as well for other problems.

All choices regarding the design issues of an evolutionary algorithm have to be made keeping in mind the other issues as well. Designing the mutation operator for example can only be done considering the representation of the individuals. We will now turn to the discussion of the parameters and operators and the possibilities we have in choosing them.

3.3.1.1 Representation

Evolutionary algorithms may incorporate any representation that is suitable for the problem. The most common representations are

- binary representation
- continuous representation
- permutations of variables
- symbolic expressions in the form of parse trees

Combinatorial problems will probably be most readily solved with permutations, whereas optimization of continuous parameters (as is the case in CFD) suggests using n-dimensional floating-point vectors, with n being the number of parameters to be optimized.

The representation must be chosen in accordance with the genetic operators. An important point is that a given range of genetic variation should lead to a respective range of differing behavior in the offspring. This implies that the step-size of the variation parameters can be chosen effectively, which depends on the representation allowing for a range of step-sizes. Furthermore, genetic algorithms use recombination to combine parts of individual solutions to generate offspring. This is facilitated significantly by a fixed-length representation. Finally, some evolutionary methods require the genetic operators to always create feasible solutions, thereby not having to sort out or repair infeasible solutions. When choosing the representation, these aspects have to be kept in mind.

3.3.1.2 Evaluation function

The evaluation function judges the quality of the generated solutions. The search can only be effective if the evaluation function and the variation operators were chosen so as to work together. A small change in a solution should also lead to a small change in its

evaluation. Otherwise, the algorithm cannot work properly. The most important criterion when designing the evaluation function is that the optimal solution should also receive the optimal evaluation! If not, there is no chance of ever finding an optimal solution. Furthermore, near-optimal solutions should also be given high evaluations.

The shape of the evaluation function depends on its representation. Some might be more amenable to evolution than others. Therefore, the representation must be kept in mind when choosing the evaluation function.

Since evaluating the individuals of a population is usually the most time-comsuming part of the optimization procedure, one might consider to accelerate the process. For instance, when running a simulation to test the individual candidates, it might not be necessary to run the whole simulation to estimate the quality of a solution. In some cases we might be able to judge whether one solution is better or worse than another after running only a part of the simulation. Here we also have to keep in mind which selection methods are being used, since it might be necessary to know exactly how much better or worse one solution is compared to another. Another possibility is to use an approximate model instead of a more exact one when running simulations. We will discuss this possibility in Chapter 6.

3.3.1.3 Population

Population size:
The larger we choose the population, the larger is the probability of finding the global optimum of a multimodal function, since more of the search space is explored. This advantage however comes with the cost of requiring a larger number of function evaluations, thereby making the search more "expensive".

Initial Population:
If any prior information about the problem is known, it should be incorporated into the initial population. For instance, one or more individuals of the initial population could be locally optimal solutions obtained by a local search algorithm. To avoid clustering of the initial population in one area of the search space, we can force the initial individuals to be some minimum distance apart from each other (depending on the representation, the distance would be measured as Euclidean distance, Hamming distance...).

3.3.1.4 Halting criterion

The halting criterion significantly influences the running time of the algorithm. The time we can afford to let the algorithm run depends on a number of things, such as the computational resources available, the urgency of finding a solution (sometimes we might require just any solution, putting up with the solution not being optimal) or simply the time we have at our disposal to finish the project.

In function of these prerequisites, we can choose different halting criteria, such as

- finding a feasible solution, as near-optimal as possible
- find a number of feasible solutions
- run the algorithm until a predefined number of steps have been performed (usually due to time and/or resource limitations)

3.3.1.5 Constraint handling

Most optimization problems involve constraints, dividing the search space into feasible and unfeasible regions. The challenge in constrained optimization lies in the treatment of unfeasible solutions. Unless we have some a priori information about the location of the global optimum, we must assume that it might lie in a feasible region close to an unfeasible region. Therefore, simply discarding unfeasible solutions might lead to a loss of information, since we might already be very close to the optimum.

The main problem is how to evaluate unfeasible solutions. Depending on the problem at hand, we might choose to abandon them completely, to "repair" them or to let them continue to participate in the evolution process, assigning them a fitness in relation to their constraint violation. Several approaches are suggested in the literature:

- Introduce a penalty function: the further a solution moves into the infeasible region, the greater the penalty. The penalty is subtracted from the solution's fitness
- Repair methods: Alter infeasible solutions, fitting them into the nearest feasible part of the search space
- Barrier methods: solutions lying in the unfeasible region are assigned a fitness of $\pm\infty$, thereby increasing the probability of discarding them
- Modify the function values in the feasible regions close to the unfeasible regions, to prevent solutions from drifting off into unfeasible regions
- Modify the objective function on the entire domain
- Convert each of the constraints into a separate objective, which has to be optimized besides the actual objectives, thereby converting the single objective optimization problem into a multiobjective optimization problem
- Overall constraint violation: Aggregate constraints to obtain one additional optimization criterion, thereby again yielding a multiobjective problem
- Favor feasible over infeasible solutions when performing selection

3.3.1.6 Generating offspring

Offspring should be generated in a way that a link between parent and offspring is maintained. The offspring should resemble the parent, otherwise we end up performing a pure random search.

We distinguish between unary and higher-order transformations. Unary transformations change a single individual to generate a new individual (e.g. mutation, inversion). Higher-order transformations combine parts from two or more individuals (e.g. recombination).

Mutation:
Mutation is achieved by changing the value of some or all parameters of a solution. Each parameter x_i subject to mutation is mutated to become x_i':

$$x_i' = x_i + \Delta x_i$$

The most common kinds of mutation procedures are

- Gaussian Mutation:
  The Gaussian distribution has the form

  $$P(x) = \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

  where σ is the standard deviation and μ is the mean. In Gaussian mutation, Δx_i is a value obtained from a Gaussian distribution with mean 0 and the standard deviation set to the mutation step size σ. Thus, $\Delta x_i = \mathcal{N}(0, \sigma)$.

- Cauchy Mutation:
  The Cauchy distribution has the form

  $$P(x) = \frac{1}{\pi}\frac{\frac{1}{2}\Gamma}{(x-\mu)^2 + (\frac{1}{2}\Gamma)^2}$$

  where Γ is the full width at half maximum and μ is the mean. When using Cauchy mutation, Δx_i is obtained from a Cauchy random distribution. For most choices of Γ, larger mutations are more likely to occur using Cauchy mutation than using Gaussian mutation, thus creating offspring more distinct from its parents.

- Uniform Mutation:
  Δx_i is a uniform random variable. Within the specified range, all values have the same probability, thus small mutations are not more likely than larger ones.

- Correlated mutations: the step sizes are represented in an $n \times n$ matrix $\mathbf{C}$. The diagonal coefficients of $\mathbf{C}$ are the variances of the mutation, the off-diagonal coefficients are the covariances. $\mathbf{C}$ is used to linearly transform a normally distributed random vector $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The mutation vector $\Delta\mathbf{x}$ is obtained as $\Delta\mathbf{x} = \sigma\sqrt{\mathbf{C}}\mathbf{z}$, where σ is the global step size.

- Correlate the mutation step size to the fitness of the solution. More successful solutions undergo less mutation.

We can choose to mutate every parameter or give each parameter a mutation probability. This implies that this parameter will only be mutated with a certain probability. We can pick a random number of individuals to be mutated/recombined or vary a random percentage of the parameters of each individual. It is also possible to fix the percentage of parameters to be mutated, thereby regulating the resemblance of the offspring with its parent(s).

We can introduce a global mutation step size that applies to all parameters or give each parameter its own step size. The latter possibility will probably be the method of choice when dealing with optimization problems where the individual parameters have different value ranges.

The mutation procedure should be designed so that it can generate any possible solution in the search space. Thereby, we obtain the possibility of exploring the entire search space. It depends on the selection method and the halting criterion which solutions are actually sampled, but the mutation procedure should provide the opportunity of generating any solution.

Permutation/Inversion:
These operators are used in combinatorial optimization. They involve the permutation of parts of the solution. Inversion is a specific form of permutation, inverting the order of some or all parameters in the solution vector. For example when solving the TSP, we can permutate the order of visiting some (or all) cities by permutating the solution vector. We can choose to change the order in one or various segments of the solution. We can also assign to every point a probability of being a starting or ending point of a segment, thereby stochastically determining the degree of permutation performed.

Recombination:
The basic recombination methods are *crossover methods* and *weighted average methods*.

Crossover denotes the process of creating an offspring by combining pieces of two or more parent solutions. These pieces can vary in length, in the extreme case the offspring receives one gene (one component of the solution) from each parent. Individual parents may also contribute varying numbers of genes. We can distinguish between *one-point crossover* and *n-point crossover*, depending on the number of crossover points.

Weighted average methods combine several parent solutions (component by component) to create an offspring. Different parent individuals can receive different weights.

We can also group the solutions into *clusters* according to their position in the search space and restrict the recombination process so that recombination takes place only within a cluster. This prevents very different solutions from entering into successful clusters. In this case we have to specify a minimum number of members in a cluster before exclusive inbreeding within the cluster takes place.

3.3.1.7 Selection

There are two phases of an algorithm in which selection takes place. The first selection picks those individuals out of the population that are to generate offspring. This is called *mating selection*. The selected individuals are copied to the *mating pool*. The second selection process takes place when determining which of the newly created individuals and which individuals of the previous generation will make up the new generation. This is called *environmental selection*.

There are almost infinite possibilities as to how selection is performed. In general, we can distinguish between deterministic and stochastic selection. Given the same population of parents and offspring, a deterministic selection method will always eliminate the same individuals, whereas a stochastic method will yield possible compositions of the next generation with certain probabilities. Deterministic selection usually converges quicker, although here again we face the danger of premature convergence.

Choosing a selection method is a trade-off between trying to improve the population and at the same time keeping a diverse set of individuals in the population so as to broaden the search space.

A few possibilities for selection methods are outlined below. Many more methods are conceivable.

Methods for environmental selection:

- Use the modified mating pool as the new population.
- Combine the sets of the modified mating pool and the old population and apply a selection method.
- Elitism: always preserve the best individuals through generations, thereby making the best solutions immortal
- *Modified Binary tournament procedure:*
  - Step 1: Randomly select two individuals out of the combined pool of parents and offspring.
  - Step 2: Discard the one with the worse fitness value, the population is thereby decreased by one.
  - Step 3: Stop if the population size is reached, else go to Step 1.

  This procedure ensures that the best solution is retained, even if not selected.

Methods for mating selection or environmental selection:

- *Roulette wheel selection* (also called proportional selection): Analogous to a roulette wheel being spun p times (p being the size of the mating pool or of the new population, respectively), each solution has a slice of the roulette wheel allocated in proportion to their fitness score. The probability of selecting the i-th individual is $P_i = F_i/\bar{F}$, where F_i denotes the fitness of the i-th individual and $\bar{F}$ is the mean fitness of the current population.
- Deterministically choose the best individuals
- Choose the i best individuals and j other individuals randomly so that $i+j=p$ with p being the size of the mating pool or the new generation, respectively.
- *Binary tournament procedure:*
  - Step 1: Randomly select two individuals out of the population.
  - Step 2: Copy the one with the better fitness value into the mating pool or the new generation.
  - Step 3: Stop if the required number is reached, else go to Step 1.
- *Random tournaments*: The best individual from a random sample survives. This process is repeated until the required number of individuals have been selected. Depending on the size of the sample, this method varies between the binary tournament (sample size two) and the deterministic method of choosing the best individuals (sample size = old population + offspring)
- *Metropolis algorithm*:
  Following the example of the Boltzmann probability distribution of energy, a solution 1 is replaced by a solution 2 with the probability

  $$P = \exp[-(E_2 - E_1)/kT]$$

  where E_1 and E_2 stand for the *energy* (corresponding to the negative fitness) of the two solutions, respectively. k refers to the Boltzmann constant and T is the current temperature. The Metropolis algorithm is used in Simulated Annealing (see Section 4.1.4), where several cycles are performed during one run, gradually decreasing the temperature. If $E_2 < E_1$, P is greater than unity. In that case it is set to unity, i.e. solution 2 is accepted. If the new solution is worse than its antecedent ($E_2 > E_1$), it is accepted with a certain probability P. Therefore, this selection scheme will sometimes accept a worse solution while always accepting a better one.
- *Linear ranking*:
  Assign to each individual a selection probability that is proportional to its rank i (worst individual has rank 0, best has rank (pop_size-1)):

  $$p(i) = \frac{2 - b + 2i(b-1)/(pop\_size - 1)}{pop\_size}$$

where b is the number of offspring to be generated by the best individual.

3.3.2 Tuning the Parameters

3.3.2.1 Tuning methods

We have several possibilities to tune the algorithm parameters to the specific problem at hand:

- trial-and-error (tuning by hand)
- mathematical analysis
- dynamic parameters

Tuning by hand is time-consuming and not guaranteed to be successful. Tuning one parameter at a time will not lead to optimal results since the parameters interact. Tuning multiple parameters at a time is unfeasible since it leads to an enormous expenditure of time. Trying all possible combinations of parameters is practically impossible.

Mathematical analysis was used by Rechenberg to optimize the standard deviation parameter. He showed that when using the [1+1] algorithm with continuous representation on a strongly convex (e.g. quadratic) bowl or a planar corridor function with a large dimensionality ($n \to \infty$) and using a Gaussian mutation with zero mean and tunable standard deviation, the best settings for the standard deviation will generate improved solutions with a probability of 0.27 and 0.18 for these two functions, respectively. Using these observations, Rechenberg formed the "one-fifth-rule": If the algorithm is generating solutions that are better than the parent with a frequency greater than one-fifth, the standard deviation should be increased. Should the frequency drop below one-fifth, decrease the standard deviation.

This result is limited in its applicability. Using this rule for other problems than the above mentioned can yield very poor results. Worth keeping in mind however is the possibility of using information about the progress of the search to vary the algorithm parameters during the search. In general, there exists very little profound theory on parameter tuning. Given we are faced with several interacting parameters to vary and usually highly nonlinear, sometimes uncontinuous functions to optimize, mathematical analysis yielding an optimal setting seems infeasible.

In general, different parameter values might be optimal at different stages of the evolution process (e.g. using large mutation steps at the beginning and smaller steps later). Therefore, dynamic parameter values suggest themselves. We can divide the methods used for dynamic parameter control into the following three categories, differentiating according to how a parameter is being altered:

- **Deterministic parameter control:**
  The strategy parameter is modified according to a deterministic rule, without using feedback from the search.
- **Adaptive parameter control:**
  The strategy parameter is modified using feedback from the search.
- **Self-adaptive parameter control:**
  The strategy parameter is encoded into the data structure (representation) of the individuals and is subject to evolution along with the solution.

Deterministic:
A parameter p is replaced by a function $p(t)$, where t can denote the number of generations, the number of fitness evaluations, the number of executed mutations or any other counter. $p(t)$ solely depends on the counter variable t, not on the quality of the solutions or any other non-deterministic aspect of the search.

Example:
Using a Gaussian mutation replacing the components x_i of an individual by

$$x_i' = x_i + \mathcal{N}(0, \sigma)$$

where $\mathcal{N}(0, \sigma)$ is a random Gaussian number with mean zero and standard deviation σ, we can vary the standard deviation according to a deterministic schedule, for example

$$\sigma(t) = 1 - 0.9\frac{t}{T}$$

where t is the generation counter and T the maximum number of generations.

Adaptive:
Adaptive schemes incorporate feedback from the search. They follow a man-made rule, but the results are not deterministic since the search itself is not deterministic. The feedback used might be a fitness criterion, diversity of population, etc.

Example:
Rechenberg's one-fifth rule to vary the standard deviation of the Gaussian mutation (enforced every n generations):

if (t **mod** $n = 0$) **then**

$$\sigma(t) = \begin{cases} \sigma(t-n)/c & \text{if } p_s > 1/5 \\ \sigma(t-n) \cdot c & \text{if } p_s < 1/5 \\ \sigma(t-n) & \text{if } p_s = 1/5 \end{cases}$$

else

$$\sigma(t) = \sigma(t-1)$$

where p_s is the frequency of successful mutations, and c is a strategy parameter (usually $0.817 \leq c \leq 1$). This rule is not deterministic, since the frequency of successful mutations is not deterministic.

Self-Adaptive:
Self-adaptive schemes are also non-deterministic, as the parameters are altered by evolution along with the solution.

Example:
Every individual $\mathbf{x}$ is assigned its own standard deviation:

$$\mathbf{x} = (x_1, \ldots, x_n, \sigma)$$

The standard deviation controls the mutation of the individual, but is itself subject to evolution. A possible variation of σ is

$$\sigma' = \sigma e^{\mathcal{N}(0,\tau_0)}$$

and thus

$$x_i' = x_i + \mathcal{N}(0, \sigma')$$

where τ_0 is a strategic parameter that is often set to $1/\sqrt{n}$. One can also introduce a separate standard deviation for every component of the individual.

Another possibility of conducting self-adaptive parameter tuning is to run a stochastic algorithm on a meta-level, whose individual solutions are themselves populations with different settings. The populations are evaluated according to their fitness (average fitness, best fitness evaluation achieved or other fitness measure). The populations with the best settings will survive. The major drawback of this hierarchical structure of stochastic algorithms is its computational complexity. In cases where evaluating the evaluation function is expensive, it is usually not feasible.

In the following, we will take a closer look at the individual parameters and some possibilities to vary them.

3.3.2.2 Representation

Depending on the representation chosen, different options suggest themselves:

- Binary representation: Allow the number of bits per parameter to vary, thereby obtaining an adaptive resolution. Also, the range of each parameter can be varied by contraction, expansion, and shifting the center of the interval.
- Binary or continuous representation: As the population converges in a variable's interval, the defined range of that variable is narrowed, thereby allowing a finer tuning.

- Example FEM: Use a coarse mesh at first to obtain solutions more quickly, when solutions converge, refine the mesh.

All variations can be performed deterministically or in an adaptive or self-adaptive manner.

3.3.2.3 Evaluation function

Varying the evaluation function only makes sense in the case where penalty functions are incorporated into the evaluation to punish infeasible or near-infeasible results. Some options include

- Vary penalties according to a deterministic schedule
- Divide one run into several cycles. One cycle ends when it has converged (according to a provided convergence criterion). The best solutions of one cycle are taken as the initial population of the next cycle. Increase the penalty pressure in every cycle, thereby changing the evaluation function.
- Increase the penalties of the constraints that have been violated by the best individuals after each run. Thereby the solutions are encouraged to satisfy these constraints in the next run.

3.3.2.4 Mutation

It often proves useful to use larger mutations of the solutions initially and smaller mutations later on in the evaluation process as the population converges. Possibilities for varying the mutation step size were already discussed in Section 3.3.2.1.

In [16], three more methods are suggested:

Cumulative Stepsize Adaptation (CSA):
The aim of this method is to reduce correlations between selected mutation steps. The mutation steps of the selected individuals are recorded and added up to construct a so-called *evolution path*. Then, the difference between the evolution path and an evolution path under random selection is reduced, since under random selection the mutation steps are uncorrelated. The global step size is adapted as follows:

- If the length of the evolution path is smaller than that of the expected evolution path under random selection, the mutation steps cancel each other out and the global step size should be reduced.
- In the other case, the mutation steps are assumed to have the same directions and can be replaced by fewer, larger steps. The step size is increased.

Covariance Matrix Adaptation (CMA):
A normally distributed vector $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ is linearly transformed by the $n \times n$ matrix $\sqrt{\mathbf{C}}$, where $\mathbf{C}$ is the *covariance matrix* of the mutation distribution. The mutation step of a solution is obtained as $\Delta x = \sigma\sqrt{C}\mathbf{z}$. The covariance matrix and the global step size are updated using the evolution paths, which are analogous to those used in the CSA. The covariance matrix is not updated as frequently as the global step size, since its computation is expensive. $\mathbf{C}$ is independent of the global step size σ. The CMA is invariant to linear transformations of the parameter space. For convex quadratic problems, $\mathbf{C}$ iteratively approaches $\mathbf{H^{-1}}$, the inverse of the Hessian matrix. The CMA works like a Quasi-Newton method in that it approximates the curvature of the function by accumulating selection information over several generations.

Reinforced Learning:
We can also use a learning algorithm to adapt step sizes. The idea is that the algorithm should "learn" when to reduce, when to maintain and when to increase the step size.

Amongst learning algorithms, we can differentiate between supervised learning and reinforcement learning. In supervised learning, examples of the desired behaviour are provided by a "teacher". The algorithm learns to imitate demonstrated behavior. For step size adaptation in optimization problems, this is not useful as the desired behavior is usually unknown. We want the algorithm to discover by itself when which actions are desirable.

Reinforcement learning is learning from interaction. The algorithm learns a *policy* for choosing an optimal *action* corresponding to its current *state* by interacting with its environment. Each action taken corresponds to a *reward*. The cumulative reward is to be maximized.

A reinforcement learning algorithm is characterized by a function mapping the action taken to the next state: $s_{t+1} = \delta(s_t, a_t)$, the reward collected for the transition to a new state: $r_t(s_t, a_t)$ and the function assigning values to possible actions to be taken at a certain state: $Q(s_t, a_t)$.

If both $\delta(s_t, a_t)$ and $r_t(s_t, a_t)$ are known, *dynamic programming* can be applied. If not, *temporal difference* methods are used. They use the received reward to update the action-value function $Q(s_t, a_t)$. We present the pseudo-code of one method, the *SARSA* algorithm, in Figure 3.2.

α denotes the learning rate. It should get smaller the more often a pair (s_t, a_t) has been visited. γ is a discount factor. If $\gamma < 1$, immediate rewards are preferred over future rewards. ϵ is the greediness parameter. The action with highest $Q(s_t, a_t)$ is chosen with a probability of $(1-\epsilon)$, a random action is chosen with the probability ϵ.

The performance of the method is highly dependant of the reward scheme.

For a more detailed description of these methods and their application to Evolution Strategies, see [16].

Outline of the algorithm:

begin
start with an initial time t:=0
initialize $Q(s_t, a_t)$ *arbitrarily*
for *each episode* **do**
Initialize s_t
Choose a_t *using* $Q(s_t, a_t)$
for *each step of episode* **do**
Take a_t, *observe* r_t, s_{t+1}
Choose a_{t+1} *using* $Q(s_{t+1}, a_{t+1})$
$Q(s_t, a_t) = Q(s_t, a_t)(1 - \alpha) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}))$
$s_t = s_{t+1}, \;\; a_t = a_{t+1}$
end

Figure 3.2: Outline of the SARSA algorithm

3.3.2.5 Selection

Some of the selection methods presented above include parameters for varying the selection pressure:

- When using the Metropolis algorithm, usually the temperature is lowered in each cycle of a run, thereby varying the selection pressure over time. This is a form of deterministic parameter control, since the temperature schedule is fixed and not subject to probability.
- Linear ranking provides a possibility for adjustment of the selective pressure by varying the parameter b. Similar methods exist for other ranking methods and tournament selection.

3.3.2.6 Population size

Some possibilities of varying the population size include

- Run a stochastic algorithm on a meta-level, where the individual solutions are populations with different population sizes.
- Resize the population as a result of the estimated variability it has. If the individual solutions are too similar, increasing the population size may enhance diversity.
- Assign a life span to each individual based on its quality upon creation. Once this life span is exceeded, the individual is removed, independent of its quality.

3.3.3 Diversity Preservation

Diversity preservation aims at preserving a wide range of solutions in the search space. In this spirit, an individual's chance of being selected is decreased the greater the density of individuals in its neighborhood.

Diversity preservation methods require a distance measure which can be defined on the genotype (the parameters of the solution), on the phenotype (the fitness achieved by the solution) with respect to the decision space, or on the phenotype with respect to the objective space. Most approaches consider the distance between two individuals as the distance between the corresponding solution parameter vectors.

3.3.3.1 Kernel methods

Kernel methods define the neighborhood of a point in terms of a so-called Kernel function K which takes the distance to another point as an argument. For each individual i, the distances $d(i,j)$ to all other individuals j are calculated and the resulting values $K(d_i)$ are summed up. The sum represents the density estimate for the individual. This density estimate can then be used to "punish" individuals in crowded regions of the search space. The fitness $F(i)$ of an individual i might be altered in the following way:

$$F_i' = \frac{F(i)}{\sum_{\substack{j=1 \\ j \neq i}}^{n} K(d(i,j))}$$

where $d(i,j)$ is a distance measure chosen in accordance with the representation.

The Kernel function is often assigned as

$$K(d) = \begin{cases} 1 - (d/\sigma_{share})^{\alpha} & \text{if } d < \sigma_{share} \\ 0 & \text{otherwise} \end{cases}$$

where σ_{share} is set to the desired size of the distance of two solutions and α is a strategy parameter. The closer a solution is to its neighbors, the more its fitness is degraded. A peak in the objective space can therefore support several solutions, whereas regions of lower fitness support less individuals. The population will therefore remain spread out across multiple extrema. This technique is also known as *niching* since individuals are competing to occupy the same decision space.

The drawbacks of this method are that it requires tuning two additional parameters (σ_{share} and α) as well as requiring the calculation of the distance between every two solutions in the population. This is infeasible for large populations.

3.3.3.2 Nearest neighbor techniques

Nearest neighbor techniques take the distance of a given point to its kth nearest neighbor into account in order to estimate the density in its neighborhood. Individuals in a crowded region of the search space are punished in a similar manner as above:

$$F_i' = \frac{F(i)}{K(d(i,k))}$$

$$K(d) = \begin{cases} 1 - (d/\sigma_{share})^\alpha & \text{if } d < \sigma_{share} \\ 1 & \text{otherwise} \end{cases}$$

3.3.3.3 Histograms

Histograms define a third category of density estimators that use a hypergrid to define neighborhoods within the space. The density around an individual is simply estimated by the number of individuals in the same box of the grid. The hypergrid can be fixed, though usually it is adapted with regard to the current population.

3.3.4 Archiving Strategies

Archiving strategies attempt at solving the problem of *deterioration* of a population: A solution contained in the population at a time t may have a lower fitness than a solution that was contained in the population at a previous time $t' < t$ but was eliminated. The question is how to prevent good solutions from getting lost. This problem does not arise when using deterministic selection, i.e. selecting the best individuals out of the combined set of the mating pool and the population. This procedure however impedes the exploration of new areas of the search space. An alternative is using an archiving strategy.

The idea is to maintain a second population, the so-called *archive*, to which promising solutions of the population are copied at each generation. Archive members may or may not be re-introduced into the evolution process, depending on the architecture of the algorithm. Different criteria are used to determine which individuals enter the archive, i.e. fitness, density information and the time that the individual has already resided in the archive. Archives are often encountered in multiobjective optimization.

3.3.5 Noise

3.3.5.1 Characterization of noise

Some optimization problems pose the problem of noise. In these problems, the objective function evaluation cannot be taken for face value, as it has errors attached to it. This can be due to measurement errors, assumptions or approximations made when generating the model of the problem. For example, if we model and calculate a problem using finite elements, we reduce the search space drastically, therefore obtaining incorrect results. All these factors can be summarized in what is refered to as "noise". We have to assume the noise to be smaller than the expected range of objective values, otherwise there is no chance of ever finding a solution close to the real optimum.

Noise can be systematic or random. For example, using a misadjusted instrument that is known to always be off by a constant value would result in systematic noise impeding the observations. Having different people read measurements from correct instruments would yield random noise, since different people are likely to misread differently and every evaluation of the measurement would yield a different result. Finite element approximations include systematic as well as random noise. The error does not change when evaluating the same point of the objective function more than once. However, since we are using numerical methods, there are a number of factors that yield random noise (round-off errors, faults in the program, ill-conditioned stiffness matrices...). If we evaluate several points that are very close to each other in the search space, we might nevertheless obtain a wide range of objective values.

Random noise can be approximated as a probability distribution (e.g. Gaussian or Cauchy). Systematic noise, when known, can directly be taken into account. When unknown, it cannot be dealt with and we have to keep in mind that the objective values of our solutions will not be exact.

If we do not have any information about the expected noise, as will be the case in most problems, we have to conduct some investigations to characterize it prior to running an algorithm on the problem. Through these investigations, we obtain a priori noise distributions. Thereby we gain information about how reliable results obtained in different regions of the search space are.

In the case of random noise, we can choose several points in the search space and evaluate them several times each to obtain an estimate of the noise distribution. When using a simulation, we should sample several points and points in their close neighborhood to see how results vary when minute changes are made to a solution. If we find a large variance in parts of the search space, these parts might be particularly unstable, meaning we should not be interested in results from these regions since we have to take into account small deviations from the exact solution in practice. For example, when manufacturing a technical piece of equipment, we cannot be sure that the measurements will be met exactly. Alternatively, a large variance will result if the simulator contains errors and

responds erratically to some inputs, in which case we should also avoid these points since the objective values are contaminated with errors whose character is unpredictable.

3.3.5.2 Fitness assignment and selection

In the presence of noise, when sampling a point we no longer obtain an objective value we can trust but rather a probability distribution describing the objective value. Points with few already sampled neighbors (i.e. points in poorly explored regions of the search space) will have a greater variance in their probability distribution. The more points we have already sampled in the vicinity of a point, the surer we can be about its objective value, i.e. its variance will generally be narrower. An exception to this rule occurs when the objective values of neighbors are inconsistent, in which case the region is unstable or the simulator fails to evaluate solutions in this region.

In many practical cases where noise is involved we will not be able to sample many points due to the computational cost of running the simulation. In these cases, we should store all visited points and their objective values. Should the objective function evaluation be cheap, yet nevertheless noisy, we should store as large a sample as affordable. Using this database, we can update the variance of the objective value probability distribution of points during the run depending on the points sampled in the neighborhood. This scheme has to be chosen according to the character of the noise.

When comparing two points, we have to take into account their respective objective value probability distributions $p(o)$. The decision whether one point is better than another can not be made solely by comparing the means of their probability distributions. For example, we might have a point a with mean m_a and a point b with mean $m_b > m_a$. If the variance of b is much higher than the variance of a, we might not want to consider it as better since if its true objective value were on the negative side of its mean, it might be a lot worse than point a. Therefore, the variance also has to be taken into account when assessing the worth of a point.

At the end of the run of the algorithm, we want to present the user with a best solution or with a collection of best solutions in the case of multiobjective optimization. To this end, we have to sort all solutions that we obtained during the run according to their worth.

The superiority of a solution a over another solution b is described by a so-called cost function $c(a, b)$. This function is a function of the difference in the true objective values of the two solutions. It has to be formulated so that a solution is "punished" for having a large variance. Therefore, it should not be symmetric but rather have a higher derivative on the negative side of the abscissa than on the positive. Given that we do not know the true objective values but their probability distributions $p(o_a)$ and $p(o_b)$, these have to be multiplied with the cost function and subsequently integrated to obtain a measure for the superiority of b over a:

$$d_b = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} p\left(o_a\right) p\left(o_b\right) c\left(o_b - o_a\right) do_a \, do_b$$

During the run of the algorithm, we need a fitness assignment to use for environmental and mating selection. Here, we need to decide between exploring unknown regions of the search space and focusing on points near previously visited solutions in order to narrow their variance. This can be achieved for example by assigning two separate fitness values, one considering the mean objective value and another the variance of the probability distribution, and then formulating an overall fitness by forming a weighted average. Alternatively, we can again formulate a cost function and form the integral as shown above.

3.3.5.3 Special cases

In general, we have to assume the noise to be unbounded, unknown and too large to be ignored. For these cases, we have presented above some approaches to characterize and handle the noise.

For the special case of problems with random noise that has a known, bounded support, *Rudolph* [22] proposes a genetic algorithm using symmetrical Beta distributions to model the noise. His algorithm is based on partially ordering the set of solutions, using the fact that the maximum value of the noise is known.

In the case of Gaussian random noise with a known variance σ^2, the uncertainty interval of the objective value can be narrowed by revisiting points. The true objective value f of a solution is in the interval

$$\left[\bar{f} - \frac{\sigma}{\sqrt{n}}\,\Phi^{-1}\left(\frac{1+\gamma}{2}\right),\ \bar{f} + \frac{\sigma}{\sqrt{n}}\,\Phi^{-1}\left(\frac{1+\gamma}{2}\right)\right]$$

with probability $\gamma > 0$. Here, $\bar{f}$ denotes the average of n objective value samples and $\Phi^{-1}(\cdot)$ is the inverse of the cumulative distribution function of the standard normal distribution [22].

If we do not treat the objective value of a solution as a probability distribution or account for its inexactness in a similar way, extra care must be taken as to which algorithm is used. In every step of any search algorithm (one generation in a multiple solution algorithm and one step in a single solution algorithm, respectively), a solution may now appear better than it is, due to noise interference. This solution is likely to be accepted if it appears to be better than the solution it is being compared to, although this step would have been rejected without noise interference.

Single solution algorithms are not well equipped to cope with this problem. Since no effort is made to narrow the uncertainty interval of the point's objective value (like resampling the point or sampling several points in its near vicinity), the algorithm might get stuck. If there are no points nearby that evaluate better than this point (which again might be the result of noise), this point will be returned as a local optimum.

Multiple solution algorithms are more suitable for random noise than single solution

algorithms. A solution has to compete for selection several times during a run. A point might get "lucky" once or twice, but in subsequent mutations, evaluations and selections, points in poor regions are very likely to get rejected after a couple of generations. A population holds more information about the landscape of the evaluation function than is contained in a single point. Observing many individuals over several generations, we gain information about the statistical average fitness in different regions of the search space. To benefit from this, it is crucial to maintain a diversity among the population (see 3.3.3).

3.3.5.4 Using gradient information

When optimizing noisy problems, care should be taken with methods approximating the gradient of the objective function. Some gradient approximation methods might be poorly suited for certain noise.

For example, if we approximate the first derivative of a function as the difference in fitness of two points over their distance (which distance measure is used is irrelevant here), we have

$$\frac{g(x_1) - g(x_2)}{|x_1 - x_2|}$$

Imposing noise, we obtain

$$\frac{(f(x_1) + h(x_1)) - (f(x_2) + h(x_2))}{|x_1 - x_2|}$$

with $h(x_1)$ and $h(x_2)$ denominating noise terms. If we are dealing with random noise, the variance of the noise acting on the gradient is the sum of the variances of $h(x_1)$ and $h(x_2)$. Depending on the variance of the noise, we may end up with a completely inaccurate estimate of the gradient. In fact, we may even obtain the wrong sign of the gradient.

On the other hand, gradient information obtained via the FEM can be a useful source of information. Since the gradient is obtained by analytically differentiating the approximation of the sought-after quantity, no additional error is made, and the convergence speed can be enhanced.

When using gradient information in the context of noise, the method of obtaining the gradient has to be chosen carefully to make sure it gives useful information rather than a random direction.

3.4 Multiobjective Optimization

In single objective optimization, the optimal solution is clearly defined; the search space is totally ordered. The objective function and the fitness function are often identical, since

we are only striving to achieve one objective. If we consider more than one objective, the search space is only partially ordered. Two solutions can be indifferent to each other. We obtain a set of optimal trade-offs, from which the user himself has to choose the single objective optimum.

Quality of a solution
A solution can be *better*, *worse* or *indifferent* to another solution with respect to the objective values. An objective vector $\mathbf{a}$ is considered *better* than another objective vector $\mathbf{b}$ ($\mathbf{a} \succ \mathbf{b}$) if no component of $\mathbf{a}$ is smaller than the corresponding component in $\mathbf{b}$ and at least one component is larger. The superior solution is said to *dominate* the inferior one.

Optimal solutions
A solution that is not dominated by any other solution in the parameter space is called an optimal solution. An optimal solution is referred to as a *Pareto-optimal*. A set of optimal solutions in the decision space $\mathbf{X}$ is called the *Pareto-optimal set*. Its image in objective space is called the *Pareto front* $\mathbf{Y}^* = \mathbf{f}(\mathbf{X}^*) \subseteq \mathbf{Y}$.

The Pareto-optimal set is a set of optimal trade-offs. A decision making process using preference information is necessary in order to select the appropriate trade-off. This is usually done by the engineer after obtaining the Pareto optimal set, however this process could also be integrated into the algorithm itself, providing it with the necessary preference information to make a choice. Some algorithms approach a multiobjective optimization problem by aggregating all objectives into a single one, thereby reducing it to a single objective optimization problem. The question of how to formulate the objective function in that case is very problem specific and is not discussed in this paper. We will investigate how to approximate the Pareto front, leaving the selection of which of the solutions to apply to the user.

In approximating the Pareto set, we have several objectives. We want to obtain a well-varied set of solutions, that is we want to maximize the diversity of the Pareto set approximation. This issue concerns selection in general (mating and environmental). The goal is to avoid a population containing mostly identical solutions (with respect to both the decision and the objective space). On the other hand, we want to minimize the distance of the generated solutions to the "true" Pareto set. This is related to mating selection and to the problem of assigning scalar fitness values in the presence of multiple optimization criteria. Furthermore, we have to prevent non-dominated solutions from getting lost, which can be achieved by elitism or archiving strategies.

When solving a multiobjective problem, we should consider the following issues:

- fitness assignment
- operators of mutation and recombination
- elitism
- selection

- diversity preservation
- constraint handling

3.4.1 Fitness Assignment

Both fitness assignment and selection must consider several objectives. Fitness can no longer be evaluated using the objective function, as is most often the case in single objective optimization, since we now face a set of objective functions. *Zitzler* [29, 30] proposes the following three types of fitness assignment for multiobjective optimization:

Aggregation-based fitness assignment

The idea is to aggregate all objectives into a single parameterized objective function. The parameters of this function are systematically varied during the optimization run in order to find a set of nondominated solutions instead of a single trade-off. For instance, some algorithms use weighted-sum aggregation, where the weights are changed during the evolution process.

Criterion-based fitness assignment

Criterion-based methods switch between the objectives during the selection phase. Each time an individual is chosen for reproduction, potentially a different objective will decide which member of the population will be copied into the mating pool. The objectives can be chosen stochastically, with the probability of an individual objective being either user-defined or randomly generated. An alternative is to use each objective equally often, so that equal parts of the population are chosen according to every objective.

Pareto-based fitness assignment

An individual's fitness is evaluated on the basis of the Pareto dominance. There are different ways of exploiting the partial order on the population:

- Dominance rank: the number of individuals by which an individual is dominated is used to determine the fitness value.
- Dominance count: the number of individuals dominated by a certain individual are taken into account.
- Dominance depth: the population is divided into several fronts and the depth reflects which front an individual belongs to.

In all of the above techniques, the fitness is related to the whole population in contrast to aggregation-based methods and criterion-based methods, which calculate an individual's fitness value independently of other individuals.

3.4.2 Mutation and Recombination

When producing offspring, it is desirable to take advantage of knowledge about prior successful variations. Ideally, we want to have a way of seeing where the algorithm is converging to and incorporate this knowledge into the offspring generating process. The problem we face in multiobjective optimization is that the solutions are not converging towards a single optimum, as there is a (sometimes infinite) number of optimal solutions.

Büche, *Milano* and *Koumoutsakos* [4] introduce an algorithm using self-organizing maps (SOM) as a method for tracking the evolution path of a multiobjective algorithm. They use the SOM to carry out recombination, mutation, and to adapt the mutation step size. The SOM is continuously trained with the current best solutions and is thus tracking the evolution path. It adapts the step size such that preference is given to areas of promising solutions in order to achieve an accelerated convergence.

We will briefly describe the underlying principles of SOM and their application to multiobjective optimization. The interested reader is refered to [3, 4].

In general, SOMs are used to approximate a distribution of points by means of a clustering process. A SOM defines a mapping of an input space $\mathbb{R}^n$ onto a lattice of m reference vectors $w \in \mathbb{R}^n$, called *neurons*. A fixed h-dimensional connectivity between the neurons is defined on the lattice. The response of the SOM to an input $x_j \in \mathbb{R}^n$ is defined as the "best matching" neuron c of all neurons i of the lattice:

$$c(x_j) = \arg\min_i \{||x_j - w_i||\}$$

In the following step, all neurons are updated so as to become closer to the input x_j:

$$w_i^{new} = w_i^{old} + \alpha H(c, w_i) \cdot (x_j - w_i), \quad i = 1, \ldots, m$$

where $\alpha \in]0, 1[$ is the learning rate and $H(c, w_i)$ is the neighborhood function assessing the distance between c and w_i. It is defined so that $H(c, c) = 1$ and $H(c, w_i) \geq 0 \; \forall \; w_i \neq c$.

Büche et al use a SOM to approximate the Pareto front. The connectivity h of the lattice is set to one dimension less than the objective space. Thereby, h corresponds to the dimension of the Pareto front. n is equal to the number of design variables. After the mating selection, the SOM is trained with the parent parameter vectors. The SOM is therefore not a reproduction of the parent generation but rather a representation of all parent generations chosen so far. The SOM is thus gathering information about the evolution of the population.

Recombination:
Recombination is performed by randomly choosing a simplex of adjacent neurons. The

offspring is obtained by picking a point within the simplex applying a uniform probability distribution.

Mutation:
To introduce further variation into the population, the offspring is also mutated. Normally distributed random numbers are added to a new point u by:

$$u_k \leftarrow u_k + \frac{\sigma}{\sqrt{n}} \mathcal{N}(0,1), \qquad k = 1, \ldots, n$$

The step size σ is defined as the Euclidean length of a randomly chosen edge of the simplex. At the beginning of the optimization, the neurons are widely distributed over the parameter space, since the initial SOM is usually chosen randomly. Thus the length of the edges is large. As the optimization advances, the difference between the individuals in the population diminishes (the distance between the neurons decreases). An additional effect of choosing the step size as an edge of the simplex is that it differs in different areas of the lattice. Thus, different areas along the Pareto front are adapted differently.

3.4.3 Elitism

As in single objective optimization, we have to consider the problem of deterioration. A solution which is part of the population at a time t might be dominated by a solution which was present at a time $t' < t$ but was rejected. This can happen when using stochastic selection methods.

As described in Section 3.3.4, we can tackle this problem by using a deterministic selection scheme on the combination of the population and the offspring, keeping all nondominated solutions in the population. We also have the option of maintaining an archive. The archive usually contains the current approximation of the Pareto set. We might also choose to keep other promising solutions. Some archives are filled partly with nondominated solutions, the remaining space being filled with random individuals. Furthermore, archive members may or may not be re-introduced into the population. As in most design issues of stochastic algorithms, there is a wide range of possibilities.

Due to limited memory capacity, we are forced to limit the number of stored solutions. Individuals have to be trimmed from the archive, for example through the use of density information or age criteria (how much time has passed since the individual entered the archive?). As soon as individuals are removed from the archive, we again face the problem of deterioration. An individual residing in the archive might be dominated by another individual that was deleted from the archive at some point. This problem should be considered when choosing the selection method.

3.4.4 Selection

As in single objective optimization, we have to perform mating selection and environmental selection. Some possible selection methods were described in Section 3.3.1.7. They also hold for multiobjective optimization.

In addition, when keeping an archive, we have to select which individuals to trim from the archive should it exceed its pre-determined size. This is more difficult since archive members are usually nondominated and therefore not comparable. As described above, we can use density and age information. The problem is to find a selection method that will ensure that during the whole run of the algorithm there existed no solution that dominates a solution contained in the final Pareto front approximation produced by the algorithm. The selection method should also take care to keep a diverse set of solutions, since we want to present the engineer with a wide range of choice. In Section 4.2.1, we will describe an archiving strategy presented by *Laumanns et al* [13], which avoids the problem of deterioration and at the same time maintains a diverse set of Pareto-optimal solutions.

Khaled [20] proposes an additional selection process. In most practical engineering optimization problems, the main time factor is the evaluation of possible solutions. Khaled proposes to screen the candidates prior to evaluation, evaluating only the solutions which pass the screening.

His work introduces a *screening module*, which decides whether an individual is likely to yield a fitness above a certain threshold. The screening module proposed uses a k-nearest neighbor technique. A sample of previous solution candidates is kept in memory. The screening module determines the k nearest neighbors of a candidate solution within the sample. If at least one of these neighbors has a fitness above the pre-determined threshold, the solution is evaluated using the expensive evaluation function. If not, it is selected with a very low probability. This is to the end of avoiding the algorithm to get stuck and simultaneously giving good solutions surrounded by bad solutions a chance to get evaluated.

Khaled proposes a sample size of 30 times the population size. The default value of k is two. It should be increased if the optimal solutions are likely to be in a very small region of the search space, surrounded by bad solutions. If the objective function is more regular or the evaluation is very expensive, k should be decreased. The default for the threshold is the fitness value of the second worst population member. Again, a higher threshold may be used for more expensive evaluation functions. The screening is started after 25% of the maximum number of evaluations have been performed, so that the sample will be representative of the search space.

One problem that might occur when using the screening module as proposed is the case where the global optimum has a very narrow basin of attraction (range of points where a specific optimum would be found if a hill-climber were to be started there). If the sample does not include any of the points in the basin of attraction, this optimum will not be

found and the algorithm will converge to a different, local optimum. The likely shape of the evaluation function should therefore be taken into account when deciding whether or not to use screening and when setting the screening parameters.

3.4.5 Diversity Preservation

As in single objective optimization, we can choose to apply Kernel methods, nearest neighbor techniques or histograms (see Section 3.3.3) to measure the density of individuals in the neighborhood of a solution. In constrast to single objective optimization, the most commonly used distance measure here is the distance between the objective vectors of two individuals and not the distance between their parameter vectors. This serves the end of obtaining a broad Pareto set.

3.4.6 Convergence Properties

A multiobjective stochastic algorithm is called globally convergent if the sequence of the Pareto front approximations it produces converges towards the true Pareto front $\mathbf{Y}^*$ as the number of generations goes to infinity [30].

The two conditions for global convergence are:

1. A mutation method that is able to produce any solution from any other
2. A selection method that is guaranteed not to cause deterioration

The first condition is relatively easy to fulfil. Laumanns algorithm (see Section 4.2.1) fulfills the second condition, while simultaneously maintaining a well-varied Pareto set approximation.

3.4.7 Quality Assessment

Unlike in single objective optimization, the quality of the results obtained from an algorithm cannot be judged unambiguously. Nondominated solutions cannot be compared amongst one another. Neither can two sets of obtained solutions be compared. Some solutions from one set may dominate solutions from the other set, some might be incomparable. There are several factors contributing to the quality of a set of solutions, such as the diversity and the distance to the real Pareto set.

Examples of unary quality measures mentioned in [30] are

- *generational distance measure*: consider the average distance of the objective vectors in the Pareto front approximation to the closest optimal objective vector

- *hypervolume measure*: consider the volume of the objective space dominated by a Pareto front approximation
- *cardinality measure*: consider the cardinality of the approximated Pareto set

Many other measures are imaginable. However, all these unary measures can not with certainty assess which one of two sets of solutions has a higher quality. Often visual evaluation will arrive at a different judgement.

Zitzler et al propose binary quality measures. They introduce the *binary ϵ-quality measure*:

Binary ϵ-quality measure:

> *Let* $\mathbf{S}, \mathbf{T} \subseteq \mathbf{X}$. *Then the binary ϵ-quality measure* $I_\epsilon(\mathbf{S}, \mathbf{T})$ *is defined as the minimum* $\epsilon \in R$ *such that any solution* $\mathbf{b} \in \mathbf{T}$ *is ϵ-dominated by at least one solution* $\mathbf{a} \in \mathbf{S}$*:*
> $I_\epsilon(\mathbf{S}, \mathbf{T}) = \min\{\epsilon \in R | \forall \mathbf{b} \in \mathbf{T} \exists \mathbf{a} \in \mathbf{S} : \mathbf{a} \prec_\epsilon \mathbf{b}\}$

There are three possible outcomes when comparing two Pareto front approximations $\mathbf{S}$ and $\mathbf{T}$:

- $I_\epsilon(\mathbf{S}, \mathbf{T}) < 1$: all solutions in $\mathbf{T}$ are dominated by a solution in $\mathbf{S}$.
- $I_\epsilon(\mathbf{S}, \mathbf{T}) = 1$ and $I_\epsilon(\mathbf{T}, \mathbf{S}) = 1$: $\mathbf{S}$ and $\mathbf{T}$ represent the same Pareto set.
- $I_\epsilon(\mathbf{S}, \mathbf{T}) > 1$ and $I_\epsilon(\mathbf{T}, \mathbf{S}) > 1$: $\mathbf{S}$ and $\mathbf{T}$ are incomparable.

This quality measure gives a factor assessing by how much one set of solutions is better than another. However, depending on the task at hand and the preference information available, other quality assessments might result more suitable.

Chapter 4

Stochastic Algorithms

In this chapter, we will introduce some of the most popular algorithms and their variations. Unfortunately, none of the authors of the algorithms provide theoretical studies justifying their specific operator and parameter choices. If available, we will report which problems the algorithm was tested on and to which other algorithms it was compared.

4.1 Single Objective Algorithms

4.1.1 Differential Evolution

Differential Evolution was developed by Storn and Price in 1995 (see [24]). We will present their original algorithm in Section 4.1.1.1. In Section 4.1.1.2, we will present a variation of the algorithm proposed by *Kučerová, Lepš* and *Zeman* [12]. In Section 5.2, we will present a hybridization of the algorithm proposed by *Rogalsky* and *Derksen* [21].

4.1.1.1 Classical Differential Evolution

The key idea of Differential Evolution is the *differential operator*, which serves the same purpose as the crossover parameter in a standard genetic algorithm, namely to exchange information between parents when creating offspring.

Storn and Price propose two different versions of Differential Evolution:

1) DE1

Let $x_i(t)$ be the i-th solution parameter vector of generation t:

$$x_i(t) = (x_{i1}(t), x_{i2}(t), \ldots, x_{in}(t)),$$

Outline of the algorithm:

begin
randomly initialize a population P
evaluate fitness of each individual p ∈ P
repeat
for every p ∈ P
generate trial solution p′
evaluate(p′)
if *fitness(p′) > fitness(p)*
then $p \leftarrow p'$
else
continue
until (*termination-condition*)
end

Figure 4.1: Outline of the Differential Evolution algorithm

where n is the length of the parameter vector (in real-value encoding, n corresponds to the number of variables of the objective function). For each vector $x_i(t)$, a so-called *trial vector* $u_i(t)$ is created by applying the differential operator:

Let Λ be a subset of $1, 2, \ldots, n$. Then for each $j \in \Lambda$ holds

$$u_{ij}(t) = x_{pj}(t) + F(x_{qj}(t) - x_{rj}(t)),$$

and for each $j \notin \Lambda$ holds

$$u_{ij}(t) = x_{ij}(t),$$

where x_{pj}, x_{qj} and x_{rj} are the j-th coordinates of three randomly chosen parameter vectors (with $p, q, r \neq i$ and $p \neq q \neq r$). F is a coefficient usually taken from the interval (0,1). It controls the amplification of the differential variation.

The size of the subset Λ determines how many parameters of each solution vector are changed. Storn and Price propose the following scheme for choosing Λ:

$$\Lambda = \{m \div n, (m+1) \div n, \ldots, (m+L-1) \div n\}$$

The starting index m is a randomly chosen integer from the interval $[0, n–1]$. L is an integer drawn from the interval $[0, n-1]$ with the probability $P(L = \nu) = (CR)^{\nu}$. $CR \in [0, 1]$ is the crossover probability and constitutes a control variable for the algorithm. Both m and L are chosen anew for each trial vector $u_i(t)$.

The hereby created individual $u_i(t)$ is compared with its parent $x_i(t)$. If it yields a higher fitness, we replace the parent with the trial vector ($x_i(t+1) = u_i(t)$). If not, the original vector is retained ($x_i(t+1) = x_i(t)$).

2) DE2

In the scheme DE2, the trial vector is generated as follows:

For each $j \in \Lambda$ holds

$$u_{ij}(t) = x_{ij}(t) + F(x_{pj}(t) - x_{qj}(t)) + \lambda(x_{bestj}(t) - x_{ij}(t)),$$

and for each $j \notin \Lambda$ holds

$$u_{ij}(t) = x_{ij}(t),$$

where x_{bestj} is the j-th coordinate of the best solution contained in generation t. λ provides a means to enhance the greediness of the scheme by incorporating the current best solution x_{best}.

Behavior of the algorithm:
The Differential Evolution algorithm only accepts a new solution if it has a higher fitness than its parent. Thus, the algorithm can get stuck in local optima. It has no means of escaping, since worse solutions are never accepted.

In DE, the crossover operator and the mutation operator have been combined to a single genetic operator, the differential operator. Mutation of a solution can thus only occur within a range defined by the population. If the population converges towards an optimum, the difference between the parameter vectors decreases. Thereby, the magnitude of the mutation is decreased. Differential Evolution thus incorporates an adaptive stepsize control. Note however that the stepsize is reduced independently of whether the algorithm is converging towards a local or a global optimum.

In order to avoid premature convergence, F should not be chosen too low. The threshold depends on the problem at hand. In general, a larger F increases the probability of escaping a local optimum while for $F > 1$ the convergence speed decreases. For DE2, F should be chosen smaller than for DE1, since two difference vectors are added to the original vector.

The crossover probability also influences the convergence. A low value speeds up convergence, again posing the problem of premature convergence. A high crossover probability turns the algorithm into a method resembling random search.

Differential Evolution incorporates no memory. If a trial vector performs better than its parent, the parent is discarded.

Storn and Price test DE1 and DE2 on a testbed consisting of nine nonlinear minimization problems. No noise was imposed on the functions. They compare the results (number of function evaluations required to find the global minimum) with those of the Annealed Nelder&Mead Strategy and the Adaptive Simulated Annealing Strategy and find their methods to be superior for these test cases.

4.1.1.2 Simplified Atavistic Differential Evolution (SADE)

This variation of Differential Evolution was proposed by [12] in order to solve high-dimensional real-valued optimization problems. It is a combination of DE1 and a genetic algorithm in that it also applies mutation. The trial vector is generated by applying either mutation or the differential operator.

Outline of the algorithm:

begin
randomly initialize a population P
evaluate fitness of each individual p ∈ P
repeat
for every p ∈ P
generate trial solution p'
evaluate(p')
apply selection
until (*termination-condition*)
end

Figure 4.2: Outline of the SADE algorithm

The differential operator is applied as in DE1. The mutation operator comes in two different forms: mutation and local mutation.

Mutation: A solution $\mathbf{x}_i$ chosen for mutation is changed as follows:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + MR(\mathbf{r} - \mathbf{x}_i(t))$$

where $\mathbf{r}$ is a new random solution and MR is the *mutation rate*.

Local mutation: When a solution $\mathbf{x}_i$ is chosen for local mutation, its parameters are altered by a random value from a given range. These changes are usually small.

Selection is applied to the double-sized population applying the modified binary tournament strategy until the original population size is re-established.

Unfortunately, the authors do not provide any explanation as to why these operators should yield enhanced performance in high-dimensional real-valued problems. They test SADE against DE2 on the Type 0-function, defined as:

$$f(\mathbf{x}) = y_0 \left(\frac{\pi}{2} - \arctan ||\mathbf{x} - \mathbf{x}_0|| \right)$$

where $\mathbf{x}_0$ is the global extreme and y_0 is a parameter controlling the height and width of the peak. This function has a single extreme on the top of a high and narrow peak, which is located in a very narrow part of the search space. As the dimension of the problem increases, it gets more challenging for the algorithms to find the optimum. According to [12], DE2 requires more than six million evaluations of the objective function for the 45-dimensional problem, while SADE solves the problem for 200 dimensions requiring just over one million evaluations.

4.1.2 Evolutionary Programming

Evolutionary Programming (EP) is the quintessential non-genetic stochastic algorithm. It works on the basis of mutation and selection but does not apply recombination of the individual solutions.

Outline of the algorithm:

begin
  initialize a population P
  evaluate(P)
  repeat
    P' = mutate(P)
    evaluate(P')
    P ← *selection (P,P')*
  until (*termination-condition*)
end

Figure 4.3: Outline of the Evolutionary Programming algorithm

The initial population is chosen at random. Each solution is replicated into a new population. The number of offspring per parent may also vary. Each of these offspring solutions are mutated according to a mutation probability distribution. Minor mutations are highly probable whereas severe mutations are less probable. Severity of mutations is often reduced as the population converges. Often self-adaptive mechanisms are used to adapt the mutation stepsize. Each offspring solution is evaluated by computing its fitness. Applying a selection procedure, the solutions to be retained for the population are determined. There is no requirement that the population size be held constant. Since no crossover takes place, there are also no constraints on the representation.

4.1.3 Evolutionary Strategies

The first *Evolution Strategy* (ES) was invented by *Rechenberg* and *Schwefel* in 1963 at the Technical University of Berlin, where they were searching for the optimal shape of bodies in a flow. Shape optimization is an example of an optimization problem where the objective function is not known in analytical form, therefore the optimization procedure relies on the intuition of the engineer. For a long time, Evolution Strategies were only popular with civil engineers [9]. The method has undergone a lot of development and we find multiple variations of it in today's literature.

ES is a genetic algorithm, since it uses mutation as well as recombination. Many different types of Evolution Strategies have been developed. They can be divided into several different types:

Types of ES:

(μ, λ) : Parent generation is not taken into account during selection, parents die off.

$(\mu + \lambda)$: Parent generation is taken into account during selection.

$(\mu/\rho, \lambda)$, ρ out of μ parents are recombined.

$(\mu/\rho + \lambda)$: Recombination usually occurs before mutation.

Population parameters:
λ: Number of offspring generated per generation
μ: Population size

The representation of the individuals is usually real-valued (no encoding). ES typically uses deterministic selection. For recombination, crossover as well as weighted average methods are used. Mutation is performed using Gaussian mutation or correlated mutations. The mutation step size is varied through self-adaptive mechanisms. The 1/5 success rule was developed by Rechenberg in the context of the (1+1)-ES. *Müller* [16] shows how to use CSA and CMA for a (μ, λ)-ES.

4.1.4 Simulated Annealing

Simulated Annealing is a single solution algorithm. It owes its name to an analogy with thermodynamics. At high temperatures, molecules of a liquid move freely with respect to one another. During cooling, the mobility is lost. Often the atoms line up to form a pure crystal which is the state of minimum energy for this system.

The essence of the process is slow cooling, allowing the molecules to redistribute themselves as they lose mobility. This is the technical definition for *annealing*. It is essential for ensuring that a low energy state will be achieved. If the liquid is cooled too quickly, irregularities are locked into the crystal structure and the energy level is higher than in a perfectly structured crystal.

The Boltzmann probability distribution

$$P(E) \sim \exp(-E/kT)$$

expresses the idea that a system in thermal equilibrium at temperature T has its energy probabilistically distributed among all different energy states E. Even at low temperatures, there is a chance of a system being in a high energy state. Therefore, there is a corresponding chance for the system to get out of a local energy minimum to find a better minimum, in the best case the global minimum. However, the lower the temperature the lower the chance of the system going uphill. k is the Boltzmann constant, a constant of nature relating temperature to energy [17].

The Boltzmann distribution was used for numerical simulations of thermodynamic systems by *Metropolis* first in 1953. The *Metropolis algorithm* states:

A system changes from an energy state E_1 to an energy state E_2 with a probability

$$p = \exp[-(E_2 - E_1)/kT]$$

Thus, if $E_2 < E_1$, the probability is greater than unity. In that case, it is set to unity, i.e. this step is always taken. An uphill step is only taken with a certain probability.

Gradient methods correspond to a rapid cooling. From the starting point, they go immediately downhill as far as possible. Thereby they find a local minimum, which is not neccessarily the global minimum. Simulated Annealing is similar to the iterated hill-climber discussed in Section 2.2.1.2. It is therefore also sometimes called *probabilistic hill-climbing*.

Outline of the Algorithm (taken from [15]):

begin
  $t \leftarrow 0$
  initialize T
  select a current point $\mathbf{v}_c$
  evaluate $\mathbf{v}_c$
  repeat
    repeat
      select $\mathbf{v}_n$ *in neighborhood of* $\mathbf{v}_c$
      if (*solution accepted*)
        then $\mathbf{v}_c \leftarrow \mathbf{v}_n$
    until (*termination-condition*)
    $T \leftarrow g(T, t)$
    $t \leftarrow t + 1$
  until(*halting-criterion*)
end

Figure 4.4: Outline of the Simulated Annealing algorithm

In applying the Metropolis algorithm to optimization, the direct analog to the energy is the objective function, which is to be minimized. There are a number of algorithm elements that the engineer has to provide. These can alter the behavior of the algorithm considerably:

- control parameter T (analog of temperature)
- annealing schedule (cooling schedule $g(T, t)$)
- mutation procedure
- policy for accepting a new solution

Below are several suggestions found in the literature.

Annealing schedules:

- Most simulated annealing algorithms follow an annealing schedule that can be summarized as follows [15]:

  – Step 1: $T \leftarrow T_{max}$
  select $\mathbf{v}_c$ at random
  – Step 2: select a point $\mathbf{v}_n$ from the neighborhood of $\mathbf{v}_c$, evaluate and decide to adopt or discard it
  repeat k_T times
  – Step 3: set $T \leftarrow rT$
  if $T \geq T_{min}$
  then goto Step 2
  else goto Step 1

  For a given number of maximum tries (e.g. bounded by the available resources), reducing T_{min} and/or increasing T_{max} (or decreasing the decay rate) reduces the number of independent attempts, but the search is more thorough during each attempt. There is a trade-off between making more independent attempts and searching more thoroughly during one attempt.

- [17]: Reduce T to $(1-\epsilon)T$ after every m moves, where ϵ/m is determined according to the specific problem.

- [17]: Choose number of total trials N, reduce T after every m moves to $T = T_0(1 - i/K)^\alpha$, where i is the number of trials realized so far, and α is a constant. Larger values of α spend more time at lower temperatures.

- [17]: After every m moves, set $T = \beta(f_1 - f_b)$, where β is an experimentally determined constant of order 1, f_1 is the smallest function value currently represented in the simplex, f_b is the best function value ever encountered.

- [11]: Choose number of total trials N, set $N_T = N^P$, then for $i = 1, \ldots, N_T$ let $T_i = W^{((N_T/C)-i)}$ and do N/N_T tests at this temperature. A value of $P = 0$ corresponds to N tests at a single temperature while $P = 1$ corresponds to 1 trial at each temperature.

Mutation procedure:

- *Press et al* [17] use a modification of the downhill simplex method. They replace the single point $\mathbf{x}$ by a simplex of $N + 1$ points.

- *Michalewicz* and *Fogel* [15] use a Gaussian distribution, where the mean is the current point and the standard deviation is set to one-sixth of the length of the variable's domain:

$$\mathbf{x} = (x_1, \ldots, x_n), \qquad x_i' \leftarrow x_i + \mathcal{N}(0, \sigma_i)$$

  The step size is adjusted over time.

Policy for accepting a new solution:

- always accept a better solution
- accept a worse solution with a certain probability p:
  - $p = \exp(f(\mathbf{x}) - f(\mathbf{x}')/T)$ [15]
  - $p = \exp(f(\mathbf{x}) - f(\mathbf{x}')/df_{first}T)$ [11], where df_{first} is the magnitude of the difference between the objective value of the starting point and the first trial at the highest temperature. It takes the place of the Boltzmann constant.

4.2 Multiobjective Algorithms

In this section, we will introduce two of the most widely known and used multiobjective optimization algorithms, SPEA2 and NSGA-II, as well as some recent approaches.

4.2.1 A Selection Algorithm for Guaranteed Convergence and Diversity

Laumanns et al [13] introduce the concept of ϵ-dominance. Their algorithm arrives at an ϵ-approximate Pareto set. These terms are defined as follows:

ϵ-Dominance:
Let $\mathbf{a}, \mathbf{b} \in \mathbf{Y}$. *Then* $\mathbf{a}$ *is said to* ϵ*- dominate* $\mathbf{b}$ *for some* $\epsilon > 0$*, denoted as* $\mathbf{a} \succ_\epsilon \mathbf{b}$*, if*
$\epsilon \cdot a_i \geq b_i \quad \forall i \in \{1, \ldots, k\}$.

ϵ-approximate Pareto front:
Let $\mathbf{Y} \subseteq R^{+^k}$ *be a set of vectors and* $\epsilon \geq 1$. *Then a set* $\mathbf{Y}_\epsilon$ *is called an* ϵ*-approximate Pareto front of* $\mathbf{Y}$*, if any vector* $\mathbf{b} \in \mathbf{Y}$ *is* ϵ*-dominated by at least one vector* $\mathbf{a} \in \mathbf{Y}_\epsilon$*, i.e.*
$\forall\, \mathbf{b} \in \mathbf{Y} \exists \mathbf{a} \in \mathbf{Y}_\epsilon : \quad \mathbf{a} \succ_\epsilon \mathbf{b}$.
The set of all ϵ*-approximate Pareto fronts of* $\mathbf{Y}$ *is denoted as* $\mathbf{P}_\epsilon(\mathbf{Y})$.

ϵ-Pareto front:

Let $\mathbf{Y} \subseteq R^{+^m}$ *be a set of vectors and* $\epsilon > 0$. *Then a set* $\mathbf{Y}_\epsilon^*$ *is called an* ϵ*-Pareto front of* $\mathbf{Y}$ *if*

1. $\mathbf{Y}_\epsilon^*$ *is an* ϵ*-approximate Pareto set of* $\mathbf{Y}$, *i.e.* $\mathbf{y}_\epsilon^* \in \mathbf{P}_\epsilon(\mathbf{Y})$, *and*
2. $\mathbf{Y}_\epsilon^*$ *contains Pareto points of* $\mathbf{Y}$ *only, i.e.* $\mathbf{Y}_\epsilon^* \subseteq \mathbf{Y}^*$.

The set of all ϵ*-Pareto fronts of* $\mathbf{Y}$ *is denoted as* $\mathbf{P}_\epsilon^*(\mathbf{Y})$.

The algorithm divides the objective space into boxes. Each objective vector uniquely belongs to one box. The algorithm maintains a set of nondominated boxes, with at most one individual in each box. The selection function takes each solution that is a possible candidate to enter the archive. The archive is then modified according to the following rules:

- If there are individuals whose box is ϵ-dominated to the box of the new solution, they are removed from the archive and the new solution is added to the archive.

- Else if there are no such solutions, but there exists an individual in the same box which is dominated by the new individual, this individual is removed and the new individual added to the archive.

- Else if there exists no individual in the same box or a dominating box, the new individual is added to the archive.

- Else, the archive remains unchanged.

This selection method can be shown to provide an ϵ-Pareto set of bounded size of all objective vectors produced by the algorithm. The algorithm guarantees that no solution better than the ones contained in the archive have been found during the run.

4.2.2 Strength Pareto Evolutionary Algorithm 2 (SPEA2)

SPEA2 was developed by *Zitzler et al* [30] and is a further development of SPEA. It uses a fixed archive size N and fills the archive with dominated individuals should there not be N nondominated individuals. Fitness is assigned incorporating density information. Mating selection is only performed on archive members.

Fitness assignment:
Each individual $\mathbf{i}$ in the population and the archive is assigned a strength value $S(\mathbf{i})$, which corresponds to the number of individuals (population members and archive members) it dominates:

$$S(\mathbf{i}) = |\{\mathbf{j} \,|\mathbf{j} \in \mathbf{P} + \mathbf{A} \wedge \mathbf{i} \succ \mathbf{j}\}|$$

A raw fitness value $R(\mathbf{i})$ of an individual is obtained by summing over the strength values of all individuals j that dominate i:

$$R(\mathbf{i}) = \sum_{\mathbf{j} \in \mathbf{P}+\mathbf{A}, \mathbf{j} \succ \mathbf{i}} S(\mathbf{j})$$

Outline of the Algorithm:

begin
  $t \leftarrow 0$
  generate initial population **P**
  create empty archive **A**
  repeat
    evaluate **P** *and* **A**
    copy nondominated individuals from **P** *to* **A**
    if (**A** exceeds N)
      then *reduce* **A**
    else
      fill **A** *with dominated individuals from* **P**
    if (*termination-condition*)
      then *stop*
    else
      perform mating selection on **A** *to fill mating pool*
      generate new population **P**
end

Figure 4.5: Outline of the SPEA2 algorithm

In this algorithm, fitness is to be minimized. A nondominated individual will therefore be assigned a raw fitness value $R(\mathbf{i})=0$.

In order to obtain a better fitness estimation in the case where most individuals do not dominate each other (and therefore have identical raw fitness values), density information is incorporated. A k-th nearest neighbor technique is applied, and the density estimate is obtained as

$$D(\mathbf{i}) = \frac{1}{\sigma_i^k + 2}$$

The fitness value is then obtained as the sum of the raw fitness value and the density:

$$F(\mathbf{i}) = R(\mathbf{i}) + D(\mathbf{i})$$

Nondominated individuals will yield a fitness value lower than one.

Environmental selection:
First, all nondominated individuals (population members as well as archive members) are copied into the archive of the next generation. If the predetermined archive size N is reached exactly, no further action is taken. If there are too few archive members, the best nondominated individuals from the population and the previous archive are used to fill up the surplus space. If N is exceeded, the archive is truncated. This is achieved by iteratively removing the individual i with the minimum distance to another individual. If there are two or more individuals with minimum distance, the second smallest distance is used as criterion.

Mating selection:
Mating selection is performed by binary tournament selection with replacement on the archive.

4.2.3 Non-Dominated Sorting Genetic Algorithm-II (NSGA-II)

NSGA-II is proposed by *Deb, Agrawal, Pratap* and *Meyarivan* [6]. Is presents an improvement to NSGA (Non-dominated Sorting Genetic Algorithm). The algorithm comprises sorting the population according to ranks of non-domination. It also defines a density estimate and a crowded comparison operator which works on the rank and the crowded distance of an individual. We will first explain the elements of the algorithm and then give an outline of a run.

The method *sort* sorts a population according to the level of non-domination of its individuals. For each solution, two values are determined: the number n_i of solutions that dominate solution i and S_i, the set of solutions that solution i dominates. Now, all solutions with a value n_i=0 are declared members of a set F_1. This set is called the first front. For every solution in the first front, we traverse its set S_i and reduce the value n_j of each contained solution j by one. All solutions whose value n_j is now zero are members of the second front, F_2.

The method *crowding-distance-assignment* assigns a distance measure to each member of the front F_i. The distance measure of a point j is obtained as the difference in objective value of the two points on either side of point j averaged over all of the objectives.

The *crowded comparison operator* $\geq_n$ defines a partial order as

$$i \geq_n j \text{ if } (i_{rank} < j_{rank}) \text{ or } ((i_{rank} = j_{rank}) \text{ and } (i_{distance} > j_{distance}))$$

where i_{rank} refers to the non-domination rank and $i_{distance}$ to the crowding distance of a solution.

In the main loop, the parent and offspring generation are combined and the resulting population is sorted. The new population is won by taking one front at a time, adding it to the population and assigning the crowding distance to its members. When the predetermined population size N is exceeded, the new population is sorted using the crowded comparison operator and the first N solutions are selected to comprise the new population. From this population, the new offspring is generated.

4.2.4 Objective Exchange Genetic Algorithm for Design Optimization (OEGADO)

Both OEGADO and OSGADO are based on the algorithm GADO, proposed by Rasheed [20]. A more detailed description and some experimental results are found in *Chafekar et*

Outline of the Algorithm (taken from [6]):

begin
initialize random population P_0
sort (P_0)
assign fitness according to non-domination level
$Q_0 =$ *generate_offspring* (P_0)
repeat
$R_t = P_t \cup Q_t$
$\mathcal{F} =$ *sort* (R_t)
while$(|P_{t+1}| < N)$
crowding-distance-assignment $(\mathcal{F}_i)$
$P_{t+1} = P_{t+1} \cup \mathcal{F}_i$
sort $(P_{t+1}, \geq_n)$
$P_{t+1} = P_{t+1}[0 : N]$
$Q_{t+1} =$ *generate_offspring* (P_{t+1})
until(*halting-criterion*)
end

Figure 4.6: Outline of the NSGA-II algorithm

al [5].

OEGADO comprises a collection of single objective GAs working concurrently to solve a multiobjective optimization problem. OEGADO runs as many GAs as there are objectives to be optimized. Each GA finds the feasible regions for its respective objective.

The GAs exchange information every pre-defined number of iterations. The GAs use *reduced models* to obtain an approximate fitness evaluation. They are won through an approximation technique. The authors use least squares approximation. These reduced models are used to rank the offspring. Only the best solutions are kept. Chafekar et al refer to this concept as *informed operators*. We will introduce this concept in detail in Section 6.2.

The GAs exchange their reduced models. Every GA calculates the approximate fitness value of an individual using the reduced models of other GAs.

For two objectives, the algorithm is described as follows [5]:

1. Both the GAs are run concurrently for the same number of iterations, each GA optimizes one of the two objectives while also forming a reduced model of it.
2. At intervals equal to twice the population size, each GA exchanges its reduced model with the other GA.
3. The conventional GA operators such as initialization (only applied in the beginning), mutation and crossover are replaced by informed operators. The IOs generate multiple children and use the reduced model to compute the approximate fitness of

these children. The best individual based on this approximate fitness is selected to be the newborn. It should be noted that the approximate fitness function used is of the other objective.

4. The true fitness function is then called to evaluate the actual fitness of the newborn corresponding to the current objective.

5. The individual is then added to the population using the replacement strategy.

6. Steps 2 through 5 are repeated until the maximum number of evaluations is exhausted.

The advantage of this approach is that the algorithm can easily be parallelized.

4.2.5 Objective Switching Genetic Algorithm for Design Optimization (OSGADO)

OSGADO uses only one single objective GA which optimizes multiple objectives in a sequential order. The algorithm switches between the individual objectives during a run.

The algorithm is described as follows [5]:

- The GA is run initially with the first objective as the measure of merit for a certain number of evaluations. The fitness of an individual is calculated based on its measure of merit and the constraint violations. Selection, crossover and mutation take place in the regular manner.

- After a certain numbers of evaluations, the GA is run for the next objective. When the evaluations for the last objective are complete, the GA switches back to the first objective.

- Step 2 is repeated till the maximum number of evaluations is reached.

Both OEGADO and OSGADO were tested on various problems and compared to NSGA-II. For most of the test cases considered, OEGADO performed better than OSGADO and at least as well as NSGA-II. However, these results cannot be generalized, as different test problems might yield different results.

Chapter 5

Hybrid Methods

5.1 Genetic Algorithm + Conjugate Gradients Method

Vicini and *Quagliarella* incorporate a gradient based optimization algorithm, namely the conjugate gradient method, into a genetic algorithm as one of the operators of the algorithm. The genetic algorithm posesses all operators of a traditional genetic algorithm and furthermore a conjugate gradient based optimization operator, which they call *hill climbing operator* (HcO). The use of the HcO depends on the fitness function being differentiable.

Offspring is created as usual using the selection, crossover and mutation operators. From these individuals, some might be selected and passed to the HcO to be improved. Afterwards, they are introduced into the new generation. The authors suggest three strategies for choosing the individuals which are fed to the HcO (for the case of single objective optimization):

1. only the fittest individual of the current generation is chosen
2. the individuals are assigned a selection probability and several are selected using the selection operator
3. several individuals are selected at random

For multiobjective optimization problems, the first strategy changes. The nondominated solutions are assigned a selection probability and a certain number are chosen at random. The multiple objectives have to be aggregated to evaluate to a scalar fitness, since the HcO can only work on scalar values.

The use of the HcO should be limited to a certain degree, since the algorithm should

not converge prematurely to a local minimum. It should merely improve some number of individuals. The authors suggest some rules for applying the HcO:

- Only use the HcO after every k generations.
- Do not let the HcO run until convergence, but rather only for one or two iterations.
- In the case where the design variables are weakly correlated, the HcO can be applied to only part of the variables.

Vicini and Quagliarella test their algorithm on one single objective and one multiobjective problem. The former is an airfoil inverse design problem, where a pressure distribution corresponding to a design point determined by the values of Mach number and angle of attack is given and the geometry of the airfoil producing this target distribution is to be found. They represent the geometry by two 5^{th} order B-spline curves (upper and lower part of the wing).

The hybrid GA is compared to two standard GAs. Results are compared by plotting the fitness value over the generations, where the hybrid methods are evaluated after 70 generations and the GAs after 100 generations, in order to consider the same number of objective function evaluations. For both GAs, results are improved when including the HcO operator. One GA achieves the best results using the HcO strategy #2 while the second works better with strategy #3. All three strategies yield better results than the GAs alone and than the gradient method by itself.

The authors furthermore show all objective function values yielded by the algorithms across 10 runs, in order to compare the scatter of the results. Again, the hybrid methods evaluate better since their results were more consistent. This result is important when evaluating the performance of an algorithm run once, instead of the average performance. In practice, there might not be enough time to run an algorithm many times so the one-run performance is important.

The multiobjective test problem consists in reducing the wave drag of an airfoil for a fixed lift coefficient and a maximum thickness while keeping the corresponding pitching moments under control. Here, hybridization yields a Pareto front with more uniformly distributed solutions of higher quality than the results obtained by a simple GA.

In summary, the hybridization of the GAs improved their results for the problems investigated. The improvement depends on the strategy of selecting which individuals are to be improved by the gradient operator, which influences the greediness of the algorithm. Depending on the optimization problem, the basic GA and the HcO parameters have to be chosen conjointly in order to work together optimally.

5.2 Differential Evolution + Downhill Simplex

Rogalsky and *Derksen* [21] offer a hybridized version of Differential Evolution, which they name HDE (Hybridized DE). They combine DE with the Downhill Simplex method. The authors argue that DE keeps the population diverse while DS quickly improves some of the solutions, letting them converge towards a local minimum. This follows the same motive we encountered in the previous section. In contrast to the Conjugate Gradient method, however, Downhill Simplex does not require any gradient information of the fitness function.

Rogalski and Derksen's version of DE differs slightly from that proposed by Storn and Price. They first form a perturbed vector according to one of the schemes DE1 or DE2. The trial vector is then formed by inheriting some parameters from the initial vector and some from the perturbed vector. The manner in which these parameters are determined differs from the classical DE scheme. The crossover constant is determined by the user as it is done in DE: $CR \in [0, 1]$. Starting at a randomly selected parameter, CR is compared to a uniformly distributed random number from the interval $[0, 1)$. Trial vector parameters are chosen from the perturbed vector until the random number generated exceeds CR, or until all parameters have been taken over. The remaining parameters are taken from the initial vector. Thus, a crossover probability of $CR = 1$ signifies that the trial vector will be a copy of the perturbed vector. The authors use $CR = 1$, thereby assuring that none of the parameters of the original vector are copied into the trial vector.

Like the HcO operator, the DS operator is only invoked every k generations in order to avoid premature convergence to a local optimum. Also it is not run until a minimum is reached, but only Nit times. In their paper, Rogalsky and Derksen propose $k = 2$ and $Nit = 4$.

The DS operator works by selecting $n+1$ solutions from the population produced by DE to form a simplex. Through reflection, the simplex is modified to improve one or several solutions. Improved solutions are then selected to pass into the next generation.

When selecting the $n+1$ solutions to form the simplex, we can either choose the $n+1$ best solutions of the population, choose solutions at random or choose some of the best and the rest at random. We have similar possibilities when choosing which of the solutions to replace by the solutions won through the usage of DS. We can replace the best, the worst or randomly chosen solutions.

The authors test HDE versus DE on three problems of airfoil design. The pressure distributions of three different airfoils are used as targets. The true solutions are known, therefore the results are compared according to how close they match the known airfoils. Unfortunately, the authors do not specify how the geometry is approximated and how the error of the results is assessed. The error is plotted over the number of flow calculations performed.

In all test cases, it was found that selecting the best $n+1$ solutions to form the simplex

yields better results than choosing solutions at random. In two of the three test cases, HDE performed significantly better than DE alone. The convergence rate was doubled. As was the case for the HcO operator, it is not unambiguous which selection strategies works best. For one of the problems, replacing the worst individual clearly was the most effective while for the other problem replacing the best and replacing solutions at random performed almost equally well. In the third test case, a Liebig pressure distribution, not all HDEs yielded better solutions than DE and only the strategy of random replacement could outperform DE, after more than 10^4 iterations.

Unfortunately, the authors do not provide any analysis as to why for two problems HDE was clearly superior in performance but did not work very well on the Liebig distribution. This case shows again that optimization problems posess different amenability to be solved using greedy algorithms. The promise of applying greedy schemes should be investigated beforehand, since the results do not always improve compared to traditional algorithms.

5.3 Genetic Algorithm + Preconditioned Descent Method

Berard et al [1] propose a hybrid approach based on a genetic algorithm and an preconditioned descent method. The gradient is approximated by an adjoint gradient method. Unlike the approaches proposed in the previous two sections, the authors use the GA to detect a close neighbor of the global minimum, and then run the descent method to converge to it quickly.

The problem investigated is the optimization of an airfoil. The airfoil is discretized as a mesh. In order to keep the number of optimization parameters down, a course mesh and a fine mesh are defined. On the course mesh, several neighboring elements of the fine mesh are agglomerated to form a patch. They are moved in the direction of an averaged normal of all the member elements of the patch. This is compensated for by a subsequent smoothing step.

The GA is run on the coarse mesh. When no significant further improvement is achieved, the best solution found so far is handed to the gradient method for further improvement.

In their paper, Berard et al only present one test case: finding the optimum shape of a wing in a 3D supersonic flow. In their example, considerable improvement was achieved on the airfoil by the gradient method. Unfortunately, the authors do not give any justification for using the parameters chosen for the genetic algorithm. Nor do they investigate if using the gradient method as an operator of the GA yields better or worse results.

5.4 Genetic Algorithm + Taylor Expansion

Berard et al [1] propose a second hybridized algorithm which uses gradient information of a few individuals to approximate the fitness of the other individuals. The algorithm aggregates the individuals into clusters, evaluating the fitness and the gradient for only one individual of each cluster exactly and approximating the fitness of the others by means of a first order Taylor expansion.

The clustering algorithm used is a K-means method. It uses Euclidean distance. The user chooses the value of K. At the start of the run, K individuals are chosen from the initial population as barycenters. Individuals are assigend to clusters according to which barycenter they are closest to. Subsequently, the center of gravity of each cluster is recomputed. This process is iterated until the clusters stabilize.

The fitness of an individual is approximated using the exactly computed fitness $J(x^*)$ and the gradient of the fitness function $\nabla J(x^*)$ of the so-called master individual x^* of the cluster. An individuals fitness is approximated as follows:

$$J(x) = J(x^*) + \nabla J(x^*)(x - x^*)$$

This approximation of the fitness is a lower estimation if the master individual is located in a convex region and is an upper estimation in the contrary case. The barycenters of the clusters are used as master individuals, since they minimize the overall error in the first order Taylor expansion. The barycenters are defined by

$$x_G = \frac{1}{K_i} \sum_{j=1}^{j=K_i} x_j \quad \text{for all } x_j \in C_i$$

The main steps of the algorithm can be summarized as

1. Run a clustering algorithm to identify clusters
2. Compute the exact fitness and gradient for barycenters of clusters
3. Update the individuals in a given cluster by means of Taylor expansion around the barycenter. Let the GA evolve to the next generation.

The cost of the algorithm is reduced significantly compared to a traditional GA, since only one exact evaluation has to made per cluster. To test their method, the authors use one 1D and one 2D function. Their results show that for comparable results, the number of function evaluations are reduced drastically by the hybrid approach. However, since the algorithm is only tested on two functions, these results are not in any way representative. For other problems, the chosen approximation might well be too imprecise. Since the method demands that the first order Taylor expansion of the fitness function be valid, it cannot be used for functions not meeting this requirement. Also for highly nonlinear problems, the results might not be as good. In this case, one should resort to other approximation methods, some of which we will discuss in Chapter 6.

Chapter 6

Approximate Models

The most crucial aspect when optimizing complex high-dimensional, multimodal problems is the computational expense of an algorithm. If we had an infinite resource of computer time, we could sample every point and thus find the global optimum of the problem. Since computer ressources are restricted, every algorithm is designed so as to search as much of the feasible design space as possible in as little time as possible.

In many engineering problems, such as finite element analysis and computational fluid dynamics, the cost of a single function evaluation can reach the order of hours, days or even weeks. Thus, it is advisable to keep the number of function evaluations as low as possible. In this chapter, we will present some possibilities of using approximate models to evaluate the objective function.

6.1 Gaussian Processes

El-Beltagy and *Keane* [8] propose using a Gaussian process approximation model (GP) instead of the full model whenever possible. The GP has the ability to provide an error bar for each prediction, thus whenever the error is too high, the exact model can be used.

Initially, the exact model is used to evaluate N randomly created test solutions $\mathbf{x}_1, \ldots, \mathbf{x}_N$. For each $\mathbf{x}_i$, the model provides a scalar output t_i. With these pairs, called the training data set $\mathcal{D}$, the initial GP is constructed.

The probability distribution $P(\mathbf{t}_N|\{\mathbf{x}_N\})$ is assumed to follow a Gaussian distribution:

$$P(\mathbf{t}_N|\mathcal{D}, \mathbf{x}_N) = \frac{1}{\sqrt{(2\pi)^N|\mathbf{C}_N|}} \exp\left[-\frac{1}{2}(\mathbf{t}_N - \mu)^T \mathbf{C}_N^{-1}(\mathbf{t}_N - \mu)\right]$$

where $\mathbf{C}_N$ denotes the covariance matrix, μ is the mean and $\{\mathbf{x}_N\}$ and $\mathbf{t}_N$ are the sets of input and output of the training data, respectively. When the data is normalized, it can

be assumed that $\mu = 0$. The joint distribution of the training outputs and the prediction $\mathbf{t}_{N+1}$ is given by

$$P(\mathbf{t}_{N+1}|\mathcal{D}, \mathbf{x}_{N+1}) = \frac{1}{\sqrt{(2\pi)^{N+1}|\mathbf{C}_{N+1}|}} \exp\left[-\frac{1}{2}\mathbf{t}_{N+1}^T \mathbf{C}_{N+1}^{-1} \mathbf{t}_{N+1}\right]$$

The covariance matrix $\mathbf{C}_N$ is calculated using a covariance function. The authors use

$$(\mathbf{C}_N)_{ij} = C(\mathbf{x}_i, \mathbf{x}_j) = \Theta_1 \exp\left[-\frac{1}{2}\sum_{l=1}^{n} \frac{(x_i^{(l)} - x_j^{(l)})}{r_l^2}\right] + \Theta_2 + \delta_{ij}\Theta_3$$

where n denotes the length of the parameter vectors. $\mathbf{C}_{N+1}$ can be expressed as follows:

$$\mathbf{C}_{N+1} = \left[\begin{array}{c|c} \mathbf{C}_N & \mathbf{k} \\ \hline \mathbf{k}^T & \kappa \end{array}\right]$$

where

$$\begin{aligned} \mathbf{k}^T &= [C(\mathbf{x}_1, \mathbf{x}_{N+1}), C(\mathbf{x}_2, \mathbf{x}_{N+1}), \ldots, C(\mathbf{x}_N, \mathbf{x}_{N+1})] \\ \kappa &= C(\mathbf{x}_{N+1}, \mathbf{x}_{N+1}) \end{aligned}$$

The predictive probability distribution for the prediction t_{N+1} is

$$\begin{aligned} P(t_{N+1}|\mathcal{D}, \mathbf{x}_{N+1}) &= \frac{P(\mathbf{t}_{N+1}|\mathcal{D}, \mathbf{x}_{N+1})}{P(\mathbf{t}_N|\mathcal{D}, \mathbf{x}_N)} \\ &= \frac{1}{\sqrt{(2\pi)\frac{|\mathbf{C}_{N+1}|}{|\mathbf{C}_N|}}} \exp\left[-\frac{1}{2}(\mathbf{t}_{N+1}^T \mathbf{C}_{N+1}^{-1} \mathbf{t}_{N+1} - \mathbf{t}_N^T \mathbf{C}_N^{-1} \mathbf{t}_N)\right] \\ &= \frac{1}{\sqrt{(2\pi)\frac{|\mathbf{C}_{N+1}|}{|\mathbf{C}_N|}}} \exp\left[-\frac{(t_{N+1} - \hat{t}_{N+1})^2}{2\sigma_{\hat{t}_{N+1}}^2}\right] \end{aligned}$$

where

$$\begin{aligned} \hat{t}_{N+1} &= \mathbf{k}^T \mathbf{C}_N^{-1} \mathbf{t}_N \\ \sigma_{\hat{t}_{N+1}}^2 &= \kappa - \mathbf{k}^T \mathbf{C}_N^{-1} \mathbf{t}_N \end{aligned}$$

For an input vector $\mathbf{x}_{N+1}$, the prediction of the output value is given by $\hat{t}_{N+1}$ and an error estimation is given by $\sigma_{\hat{t}_{N+1}}^2$.

The GP has several *hyperparameters*:

$$\Theta = log(\Theta_1, \Theta_2, \Theta_3, \mathbf{r})$$

The hyperparameters are defined as the log of the variables used in the covariance function in order to guarantee their values to be positive. Θ_1 controls the overall vertical scale relative to the mean, Θ_2 sets the bias of the correlation, Θ_2 sets the noise level and $\mathbf{r}$ is a distance measure. The vector $\mathbf{r}$ has the same dimension as the parameter vectors, therefore each of its coefficients r_l is a distance measure for one input dimension. For irrelevant input, the corresponding r_l is large and that input will not have a large influence on the model. This property is termed automatic relevance determination.

The posterior probability of the hyperparameters is

$$P(\Theta, \mathcal{D}) = \frac{P(\mathbf{t}_N|\{\mathbf{x}_N\}, \Theta)P(\Theta|\{\mathbf{x}_N\})}{P(\mathbf{t}_N|\{\mathbf{x}_N\})}$$

To determine the maximum a posteriori estimate for Θ, the logarithm of $P(\Theta, \mathcal{D})$ is maximized:

$$\begin{aligned}
\mathcal{L} &= \ln P(\mathbf{t}_N|\{\mathbf{x}_N\}, \Theta) + \ln P(\Theta|\{\mathbf{x}_N\}) - \ln P(\mathbf{t}_N|\{\mathbf{x}_N\}) \\
&= \ln P(\mathbf{t}_N|\{\mathbf{x}_N\}, \Theta) + \ln P(\Theta) + const \\
&= -\frac{1}{2}\log|\mathbf{C}_N| - \frac{1}{2}\mathbf{t}_N^T\mathbf{C}_N^{-1}\mathbf{t}_N - \frac{N}{2}\log 2\pi + \ln P(\Theta) + const
\end{aligned}$$

The maximization of this probability can be done in various ways. The authors use a conjugate gradient optimizer.

After the maximum a posteriori value of $\boldsymbol{\Theta}$ is determined, the final covariance matrix is calculated. Now the GP is assembled and can be used as an approximate model. For every input vector $\mathbf{x}$, we obtain an estimated output value $\hat{t}$ and an error estimate σ^2. If the error is not greater than a predefined value, the estimate obtained via the GP is used to evaluate the solution. If the error is too large, the expensive model is consulted.

During the run of the optimization algorithm, the GP is expanded when a point is evaluated using the expensive model. The hyperparameters are only re-optimized when the number of added points exceeds 40% of the number of points that were used for the last model update. This strategy is based on the assumption that expanding the model does not significantly change the values of the hyperparameters.

The model is updated by calculating the new inverse of the covariance matrix. This is done using inversion by partitioning [17]. In general, when adding M new points to the model, the partitioned inverse equation is

$$\tilde{\mathbf{C}}_L^{-1} = \begin{bmatrix} \mathbf{M} & \mathbf{K} \\ \hat{\mathbf{K}}^{\mathbf{T}} & \hat{\mathbf{V}} \end{bmatrix}$$

where

$$\begin{aligned}\hat{\mathbf{V}} &= (\mathbf{V} - \mathbf{K}^T\mathbf{C}_N^{-1}\mathbf{K})^{-1}\\ \hat{\mathbf{K}} &= -\mathbf{C}_N^{-1}\mathbf{K}\hat{\mathbf{V}}\\ \mathbf{M} &= \mathbf{C}_N^{-1} + \hat{\mathbf{K}}\mathbf{K}^T\mathbf{C}_N^{-1}\end{aligned}$$

and

$$\mathbf{K} = \begin{bmatrix} C(\mathbf{x}_1, \mathbf{x}_{N+1}) & \dots & C(\mathbf{x}_1, \mathbf{x}_{N+M}) \\ \vdots & \ddots & \vdots \\ C(\mathbf{x}_N, \mathbf{x}_{N+1}) & \dots & C(\mathbf{x}_N, \mathbf{x}_{N+M}) \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} C(\mathbf{x}_{N+1}, \mathbf{x}_{N+1}) & \dots & C(\mathbf{x}_{N+1}, \mathbf{x}_{N+M}) \\ \vdots & \ddots & \vdots \\ C(\mathbf{x}_{N+1}, \mathbf{x}_{N+M}) & \dots & C(\mathbf{x}_{N+M}, \mathbf{x}_{N+M}) \end{bmatrix}$$

The procedure of the algorithm can be outlined as follows:

Outline of the algorithm:

input *maxeval, maxstdtol*
begin
Random population initialization
Evaluation of Np individuals
Nacc=Np
OldNacc=Nacc
Construct initial Gaussian process
while *Nacc < maxeval*
Apply genetic operators
for *i*=1 to *Np*
$stdtol = maxstdtol\frac{maxeval-Nacc}{maxeval-Np}$
if $(\sigma(\mathbf{p}_i) > stdtol)$
evaluate $\mathbf{p}_i$ *using expensive model*
expand the GP to include $\mathbf{p}_i$
Nacc=Nacc+1
else *evaluate* $\mathbf{p}_i$ *using the GP*
if(*OldNacc==Nacc*)
maxstdtol = maxstdtol/2
OldNacc=Nacc
end

Figure 6.1: GA using Gaussian Process approximation model

The parameters have the following meaning:

maxeval	maximum number of affordable expensive model evaluations

maxstdtol	maximum allowable tolerance on the prediction uncertainty
stdtol	currently allowable tolerance (is initially maxstdtol, then decreases linearly to zero)

The algorithm works by first randomly initializing a population of size *Np*. These individuals are evaluated using the expensive model. With these results, the initial GP is constructed. The population is altered using the genetic operators. For each individual, we calculate the predicted standard deviation σ. If the value of σ does not exceed the tolerance value, the individual is evaluated using the GP. Should the predicted error be too large, the expensive model is consulted and the GP is expanded by this point. If no individual of a new population is evaluated with the expensive model,the allowable tolerance is tightened by a factor of two. This prevents the algorithm from exclusively using the GP. The algorithm runs until the maximum number of expensive model evaluations is exhausted.

6.2 Informed Operators and Quadratic Least Squares Approximation

Informed Operators (IO) are proposed by *Rasheed et al* [18, 19]. They replace the conventional genetic operators with operators that are guided by a reduced model in order to enhance the algorithm speed. They propose four types of informed operators:

- **Informed initialization:** In order to form the initial population, uniformly random individuals are created and the best are selected using the reduced model.
- **Informed mutation:** Several random mutations of a point are generated by randomly choosing from several mutation methods and parameters. The best offspring is selected using the reduced model.
- **Informed crossover:** From two parents, several individuals are created by using different randomly chosen crossover methods and parameters. Informed mutation is applied to every potential offspring. The best offspring is selected using the reduced model.
- **Informed guided crossover:** Guided crossover is introduced in [20]. It involves first selecting several individuals to be candidates for the first parent in the crossover. For each potential first parent, a mate is selected. Several random individuals are created. The best offspring is selected using the reduced model.

For each of these operators, the authors suggest a number of individuals to create resulting in a certain number of reduced model calls.

During the run of the algorithm, a sample of previously evaluated points is kept. The sample may contain all points or a selection of points. The sample is divided into clusters.

The algorithm starts with one cluster and introduces an additional cluster every specific number of iterations. Every new point entering the sample either becomes part of an already existing cluster or forms a new cluster, if it is time to create a new cluster. Point are allocated to the cluster whose center is the closest measured in Euclidean distance of the decision vector.

In order to evaluate the fitness of an individual using the reduced model, the authors suggest several different techniques. We will only introduce the quadratic least squares approximation here, since it is a very quick way of forming an approximation, in fact more than an order of magnitude faster than the other methods suggested.

Two types of approximation functions are defined. A separate approximation function is formed for the fitness and for the sum of constraint violations.

- **Global approximation functions**
  The two global approximation functions are based on all evaluable points in the sample. They are quadratic function of the form

  $$\hat{F}(\bar{X}) = a_0 + \sum_{i=1}^{n} a_i x_i + \sum_{i=1,j=i}^{n,n} a_{ij} x_i x_j$$

  where n is the dimension of the search space and x_i is the design variable number i. The coefficients a_i are found using a least square fitting routine from [17].

- **Cluster approximation functions**
  Cluster approximation functions are formed analogously as described above, except that the functions are only formed for clusters which have a sufficient number of evaluable points.

When evaluating a point, it is first assigned to a cluster. If this cluster already posesses cluster approximation functions, these will be used to approximate the fitness and the constraint violations. Otherwise, the global approximation functions are used. In the first half of the run of the algorithm, a point is evaluated without looking if it is likely to be feasible, infeasible or unevaluable. In the second half, we examine the nearest neighbor of the new point to assess which type of point we are likely to have. If the point is guessed to be unevaluable, it is not evaluated. For points guessed to be feasible, the sum of constraint violation is set to zero.

Bibliography

[1] Berard, Désidéri, Habbal, Janka, and Oulladji. Experiments with hybridized genetic algorithms in aerodynamics. In *International Congress on Evolutionary Methods for Design, Optimisation and Control with Applications to Industrial Problems*. CIMNE, Barcelona, Spain, 2003.

[2] Frédéric Bonnans, Charles Gilbert, Claude Lemarcéchal, and Claudia Sagastizábal. *Numerical Optimization*. Springer Verlag Berlin, 2003.

[3] D. Büche, G. Guidati, P. Stoll, and P. Koumoutsakos. Self-organizing maps for pareto optimization of airfoils. In *Seventh International Conference on Parallel Problem Solving from Nature (PPSN VII)*, Granada, Spain, 2002. Springer Verlag.

[4] D. Büche, M. Milano, and P. Koumoutsakos. Self-organizing maps for multiobjective optimization. In *Workshop Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 152–155. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[5] Deepti Chafekar, Jiang Xuan, and Khaled Rasheed. Constrained multi-objective optimization using steady state genetic algorithms.

[6] Kalyanmoy Deb, Samir Agrawal, Amrit Pratab, and T. Meyarivan. A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. In Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyne Lutton, J. J. Merelo, and Hans-Paul Schwefel, editors, *Proceedings of the Parallel Problem Solving from Nature VI Conference*, pages 849–858, Paris, France, 2000. Springer. Lecture Notes in Computer Science No. 1917.

[7] Stefan Droste, Thomas Jansen, and Ingo Wegener. Perhaps not a free lunch but at least a free appetizer. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the First Genetic and Evolutionary Computation Conference(GECCO '99)*, pages 833–839, San Francisco CA, 13–17 1999. Morgan Kaufmann Publishers, Inc.

[8] Mohammed A. El-Beltagy and Andy J. Keane. Evolutionary optimization for computationally expensive problems using Gaussian processes. In *Proceedings of the*

International Conference on Artificial Intelligence IC-AI, pages 708–714. CSREA Press, 2001. citeseer.nj.nec.com/460461.html.

[9] Jörg Heitkötter and David Beasley. The Hitch-Hiker's Guide to Evolutionary Programming. http://www.etsimo.uniovi.es/ftp/pub/EC/FAQ/www/, 1993-1998. FAQ for comp.ai.genetic.

[10] Grzegorz Kaczmarczyk. Downhill simplex method for many (∼20) dimensions. Institute of Theoretical Physics and Astrophysics, University of Gdansk, http://paula.univ.gda.pl/∼dokgrk/simplex.html.

[11] Andy. J. Keane. A brief comparison of some evolutionary optimization methods, 1996. citeseer.nj.nec.com/keane96brief.html.

[12] Kučerová, Lepš, and Zeman. Applying genetic algorithms to several problems of engineering practice. In *International Congress on Evolutionary Methods for Design, Optimisation and Control with Applications to Industrial Problems*. CIMNE, Barcelona, Spain, 2003.

[13] Marco Laumanns, Lothar Thiele, and Eckart Zitzler. Archiving with guaranteed convergence and diversity in multi-objective optimization, 2002.

[14] William G. Macready and David H. Wolpert. What makes an optimization problem hard? Technical Report SFI-TR-95-05-046, Santa Fe, NM, February 1996.

[15] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer Verlag Berlin, 2002.

[16] Sibylle D. Müller. *Bio-inspired optimization algorithms for engineering applications*. PhD thesis, ETH Zürich, 2002. http://e-collection.ethbib.ethz.ch/show?type=diss&nr=14719.

[17] William H. Press et al. *Numerical Recipes in C , The Art of Scientific Computing*. Cambridge University Press, 1997.

[18] Khaled Rasheed, Swarop Vattam, and xiao Ni. Comparison of methods for using reduced models to speed up design optimization, 2002.

[19] Khaled Rasheed, xiao Ni, and Swarop Vattam. Comparison of methods for developing dynamic reduced models for design optimization. In *Proceedings of the Congress on Evolutionary Computation (CEC 2002)*, 2002.

[20] Khaled Mohamed Rasheed. *GADO: A Genetic Algorithm for Continuous Design Optimization*. PhD thesis, Rutgers, The State University of New Jersey, 1998. http://webster.cs.uga.edu/ khaled/thesis.ps.

[21] T. Rogalsky and R. W. Derksen. Hybridization of differential evolution for aerodynamic design. In *Proceedings of the 8th Annual Conference of the Computational Fluid Dynamics Society of Canada*, pages 729–736, June 11-13 2000. citeseer.nj.nec.com/315773.html.

[22] Guenter Rudolph. A partial order approach to noisy fitness functions. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 318–325, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 2001. IEEE Press.

[23] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. http://www-2.cs.cmu.edu/∼jrs/jrspapers.html, 1994.

[24] Rainer Storn and Kenneth Price. Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous space. Technical Report TR-95-012, International Computer Science Institute, Berkeley, CA, USA, March 1995. ftp://ftp.icsi.berkeley.edu/pub/techreports/1995/tr-95-012.pdf.

[25] Malcolm A. Strens. Evolutionary mcmc sampling and optimization in discrete spaces. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML)*, 2003.

[26] A. Vicini and D. Quagliarella. Airfoil and wing design through hybrid optimization strategies. *AIAA Journal*, 37(5):634–641, May 1999.

[27] Eric Weisstein. Eric Weisstein's World of Mathematics. http://mathworld.wolfram.com.

[28] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997. citeseer.nj.nec.com/wolpert96no.html.

[29] Eckart Zitzler. Evolutionary algorithms for multiobjective optimization. In *Evolutionary Methods for Design, Optimisation and Control*, pages 19–26. CIMNE, Barcelona, Spain, 2002. ftp://ftp.tik.ee.ethz.ch/pub/people/zitzler/Zitz2002a.pdf.

[30] Eckart Zitzler, Marco Laumanns, and Stefan Bleuler. A tutorial on evolutionary multiobjective optimization. In *Workshop on Multiple Objective Metaheuristics (MOMH)*. Springer Verlag Berlin, 2003. ftp://ftp.tik.ee.ethz.ch/pub/people/zitzler/ZLB2003a.pdf.