EALib: A C++ class library for evolutionary algorithms

June 27, 2013

Institut für Neuroinformatik, Ruhr-Universität Bochum

http://www.neuroinformatik.ruhr-uni-bochum.de/ini/PEOPLE/kreutz/top.html
Martin.Kreutz@neuroinformatik.ruhr-uni-bochum.de

The EALib C++ library is part of the Shark library. It contains classes for the implementation of evolutionary algorithms and related techniques. The library also includes example programs, which can be used as templates for own programs to do evolutionary optimization. The whole package is intended to be used in evolutionary algorithms research. Therefore, the main emphasis has been put on providing as much flexibility and extendability as possible. Efficiency has been another major design issue, but has been sacrificed where it conflicts with flexibility. This documentation gives a short overview of how evolutionary algorithms work, the most important operators, the programming interface, and examples illustrating the usage of the EALib classes.

CONTENTS

1 EVOLUTIONARY ALGORITHMS – CLASS LIBRARY

1.1 Introduction

Evolutionary Algorithms (EA) are direct, probabilistic search and optimization algorithms. They are inspired by principles of neo-Darwinean evolution theory. In particular, they usually maintain a *population* of candidate solutions (i.e, points in the search space). The main variation operators are *mutation* and *recombination*. During one search cycle, or *generation*, the members of the population are ranked according to their associated quality or *fitness* which determines the probability of being reproduced in the next generation. This process is called (environmental) *selection*. The definiton of the fitness is problem dependent and represents the goal of optimization. Another crucial point in the successful application of evolutionary algorithms is the appropriate choice of the *coding*. Although the right combination of the evolutionary operators and the coding depends on the specific problem or at least the problem domain, the basic structure of an evolutionary algorithm remains, in principle, the same (see Algorithm 1.1). In order to simplify the formulation and implementation of specific EAs we have developed a class library which contains a set of data structures and evolutionary operators acting upon them. In the remainder of this chapter, basic components of this library are introduced and explained with the help of some simple examples.

$t := 0;$
$initialize(P(t));$
$evaluate(P(t));$
while not $terminate(P(t))$ **do**
    $P(t+1) := select(P(t));$
    $recombine(P(t+1));$
    $mutate(P(t+1));$
    $evaluate(P(t+1));$
    $t := t+1;$
od

Algorithm 1.1: Outline of an Evolutionary Algorithm. $P(t)$ denotes the population at generation t.

1.2 Mainstream Paradigms

1.2.1 Genetic Algorithms

Genetic algorithms (GAs) in their classical form were invented by John Holland, a computer scientist and psychologist at the University of Michigan, in the 1970s to mimic features of natural evolution [21]. However, earlier predecessors of these type of algorithms were developed by Fraser, a biologist, in the late 1950s. He started with the intention to simulate natural evolution in order to improve the understanding of evolution from the biological point of view rather than having practical applications in mind.

The main difference between GAs and other evolutionary paradigms like evolution strategy (ES) or evolutionary programming (EP) lies in the representation of the *genotype*. GAs operate on symbol strings (usually on bitstrings) of a fixed length, that is the *phenotype* has to be encoded into this string. For pseudoboolean fitness functions this represention needs no special encoding procedure. For general fitness functions, however, the coding plays a significant role and has an important impact on the applicability or at least on the performance of a GA. In the domain of continuous parameter optimization, for example, the bitstring has to be decoded to a vector of continuous values. Two encoding schemes are commonly used in this domain, the binary code and the Gray code. For tasks where the size and structure of the phenotype is no longer fixed, like in the field of structure optimization of neural networks, more complicated encoding schemes have to be employed.

1.2.2 Evolution Strategies

Evolution Strategies (ES) were developed in the 1960s at the Technical University of Berlin by Bienert, Rechenberg and Schwefel. Although GAs and ESs share some basic concepts (adopted from nature) they were developed independently from each other. The first version of ES worked with only one parent and one offspring individual and therefore did not incorporate the population principle. An outline of this first (1+1)-ES is shown in Algorithm 1.2.

In later versions of ES the population concept were introduced, which led to (μ+λ)-ES and (μ,λ)-ES (for details see Section 1.4.1.2). First applications of ES dealt with hydrodynamical problems like shape optimization of a bended pipe, drag minimization of a joint plate, and structure optimization of a two-phase flashing nozzle. The first two applications dealt with parameter optimization, whereas the latter one was a first example of structure optimization. However, most applications of ES come from the domain of continuous parameter optimization, in which ES have proven to be very successful. The advantage compared to other evolutionary algorithms like GAs seems to abe that the encoding, a vector of floating point numbers combined with normally distributed mutation steps (and an adaptation of the mutation distribution) is well suited for parameter optimisation problems.

1.3 Data Structures

Evolutionary algorithms are formulated in terms of populations, individuals, chromosomes, and genetic operators working on them. In this chapter we introduce the basic data structures which are used to represent populations and individuals of arbitrary structure and size.

$t := 0$;
$initialize(parent)$;
$evaluate(parent)$;
while not $terminate(t)$ **do**
    $offspring := mutate(parent)$;
    $evaluate(offspring)$;
    $parent := best(parent, offspring)$;
    $t := t + 1$;
od

Algorithm 1.2: Outline of the (1+1)-Evolution Strategy. The offspring individual is created from its parent by applying the mutation operator to the genome of the parent. The resulting individual is evaluated and compared to its parent. The better one of both survives to become the parent of the next generation while the other one is discarded.

1.3.1 Basic Data Types

Many of the higher level data types are built up from simpler types. The abstract data type `vector< type >` implements a simple sequence of items of any type and is realized via *C++* templates. Instances of class `vector< type >` can be used wherever a *C++* `type*` string can be used. However, they differ from `type*` in several aspects: parameter passing by value and assignment works properly (i.e., the value is passed or assigned and not a pointer to the values) and the subscript operator [ ] may perform a range check at run–time (this is implemented by the exception mechanism of *C++*). Furthermore, instances of `vector< type >` can be resized dynamically. Chromosomes, individuals, and populations which are essentially sequences of items are derived from `vector`. The data type `vector` as well as other basic data types are defined in the *Standard Template Library* (see [25] for details).

1.3.2 Genes and Chromosomes

The representation (*genome*) of an individual consists of a variable number of sequences (*chomosomes*) of variable length. These sequences may contain binary, integer, continuous values, or any other type of values (*alleles*). The base class `Chromosome` defines the common properties of all types of chromosomes. Furthermore, it declares genetic operators which can be applied to any sequence of items such as crossover, inversion, duplication, rotation, translation. The more specific operators like initialisation, mutation, and special forms of recombination, are defined in the derived template class `ChromosomeT< type >`. Some general operators, like input from streams and output to streams, depend on the special `type` of a chromosome and may not be defined for that type. However, they are declared in the base class `Chromosome` to be accessible by any derived class. Unless they are overloaded by the derived classes a call of these operators may cause the exception *undefined operator*.

1.3.2.1 Binary Valued Alleles

Chromosomes with binary valued alleles are essentially strings of Boolean values. The class `ChromosomeT< bool >` is derived from class `Chromosome` and class `vector< bool >`. All operators which are defined for class `vector< bool >` such as assignment, subscript, test for equality, and change of size are available for class `ChromosomeT< bool >` in the same way. Instances of class `ChromosomeT< bool >` can be used whereever a *C++* `bool *` string can be used. An example is shown in Example 1.1 (a). Furthermore, some genetic operators specific for strings of boolean values like flipping particular alleles or interpreting the genome as a binary encoded floating point number are defined in this class.

1.3.2.2 Integer Valued Alleles

Analogously, the class `Chromosome< int >` is derived from class `Chromosome` and class `vector< int >`. Alleles of an instance of `ChromosomeT< int >` may contain integer values as well as symbols (see Example 1.1 (b)).

1.3.2.3 Continuous Valued Alleles

The class `ChromosomeT< double >` is specially designed for continuous parameter optimisation problems. All variants of evolution strategies rely on this data type. A variety of mutation operators, mainly based on normally distributed random numbers, and special forms of recombination are implemented in this class (see Example 1.1 (c)).

1.3.3 Individuals

Individuals contain a sequence of chromosomes, the *genome*. Size and structure of the genome is determined by the number of chromosomes and their types (see Figure 1.1). Furthermore, an individual contains information about its *fitness*, the corresponding value of the objective function (if any), a flag which signals whether the genome represents a feasible solution, and some internal variables.

If the genome of an individual contains chromosomes of arbitrary types there is a problem to access certain members of the genome, for example via the subscript operator [ ]. The type of these members is not fixed in advance, that is they are *polymorphic*. A possibility to handle this problem is to cast the members to their respective types. The standard method in *C++* is to use the operator `dynamic_cast< type >( )` which tests if the requested conversion is possible and casts the argument to the given type.

The subscript operator [ ] applied to an individual returns the chromosome specified by the index. A short example is outlined in Example 1.4.

If an individual has only chromosomes of a single type, we can use the template class `IndividualT` and the casts become unnecessary. This is shown in Example 1.3.

```
            ⋮
#include <EALib/ChromosomeT.h>          //  declarations
            ⋮
ChromosomeT< bool > chrom( 10 ); //  define chromosome with 10 alleles
chrom.initialize( );             //  initialize alleles with random values

for( int i = 0; i < chrom.size( ); i++ ) {
    cout << "allele " << i << " = ";
    cout << chrom[ i ];          //  send ith allele to standard output
    cout << endl;
}
            ⋮
```

(a) definition of a binary valued chromosome and some operations on it

```
            ⋮
#include <EALib/ChromosomeT.h>          //  declarations
            ⋮
ChromosomeT< int > chrom( 3 );   //  define chromosome with 3 alleles
            ⋮
```

(b) definition of an integer valued chromosome

```
            ⋮
#include <EALib/ChromosomeT.h>         //  declarations
            ⋮
ChromosomeT< double > chrom( 20, 1.2 );  //  define chromosome with 20
                                         //  alleles with initial value 1.2
            ⋮
```

(c) definition of a continuous valued chromosome

Example 1.1: Definitions of chromosomes of different types

```
                ⋮
//
//  define an individual with two chromosomes
//
Individual indiv( ChromosomeT< int  >( 5 ),
                  ChromosomeT< char >( 8 ) );

//
//  dynamic type casting
//
dynamic_cast< ChromosomeT< int >& >( indiv[ 0 ] ).initialize( 0, 9 );
                ⋮
```

Example 1.2: Dynamic type casting of chromosomes.

```
                ⋮
//
//  define an individual with two chromosomes
//
IndividualT< int > indiv( ChromosomeT< int  >( 5 ),
                          ChromosomeT< int >( 8 ) );

//
//  Initialization
//
indiv[ 0 ].initialize( 0, 9 );
indiv[ 1 ].initialize( 4, 42 );
                ⋮
```

Example 1.3: Individuals having a single type of chromosomes are easier to handle.

1.3.4 Populations

A population represents a collection of individuals. Individuals may be sorted in a population either in ascending or descending order with respect to their fitness. For class `Population` genetic operators like selection and replacement are defined. The [ ] applied to a population returns the individual specified by the index. For a short example see Example 1.5.

For populations consisting of individuals from the template class `IndividualT`, the convenient template class `PopulationT` has to be used and we get rid of the cast operations. This can be seen in most examples at the end of this manual.

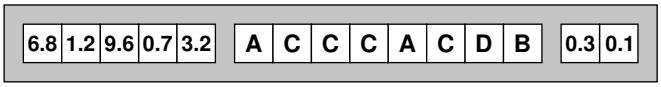

Figure 1.1: Example for a representation of the genome of an individual consisting of three chromosomes.

1.4 Genetic Operators

1.4.1 Selection

Selection occurs two times in the evolutionary loop. First, in order to generate offsprings, parents must be selected from the current population (mating selection). Second, the new

```
            ⋮
#include <EALib/Individual.h>  // declarations
            ⋮
//
//  define an individual with two chromosomes
//
Individual indiv( ChromosomeT< bool   >( 80 ),
                  ChromosomeT< double >( 10 ) );

//
//  define range of values
//
Interval range( -1.5, +3.0 );

//
//  initialize alleles of chromosome #0
//
dynamic_cast< ChromosomeT< bool >& >
    ( indiv[ 0 ] ).initialize( );

//
//  decode the binary values in chromosome #0
//  to the floating point chromosome #1
//
dynamic_cast< ChromosomeT< double >& >
    ( indiv[ 1 ] ).decodeBinary( indiv[ 0 ], range, 10 );
            ⋮
```

Example 1.4: Definition of a sample individual and some operations on its genome

```
            ⋮
#include <EALib/Population.h>  //  declarations
            ⋮
//
//  define population size
//
const unsigned PopSize = 20;

//
//  define population with three chromosomes per individual
//
Population pop( PopSize, ChromosomeT< char   >( 3 ),
                         ChromosomeT< int    >( 5 ),
                         ChromosomeT< double >( 8 ) );

//
//  initialize alleles of chromosomes
//
for( unsigned i = 0; i < pop.size( ); ++i ) {
    dynamic_cast( ChromosomeT< char >& >
        ( pop[ i ][ 0 ] ).initialize( 'A', 'Z' );
    dynamic_cast( ChromosomeT< int >& >
        ( pop[ i ][ 1 ] ).initialize( -13,  10 );
    dynamic_cast( ChromosomeT< double >& >
        ( pop[ i ][ 2 ] ).initialize( 1.5, 4.5 );
}
            ⋮
```

Example 1.5: Definition of a sample population with 20 individuals.

parent population has to be selected from the offspring and the previous parents. A number of selection operators were proposed, which usually base the chance of selection of particular individuals on their fitness values or their rank in the population, respectively.

1.4.1.1 General Schemes

Extinctiveness With respect to the selection probability of each individual, selection schemes divide into two major groups: *preservative* and *extinctive* selection. In the preservative version each individual is assigned a nonzero selection probability, hence each individual has (in principle) the chance to reproduce, whereas in extinctive selection schemes some individuals are definitely discarded, that is their associated selection probability equals to zero. A typical example is the (μ, λ)–Selection (see Section 1.4.1.2). More details and a

comparison of the different selection schemes can be found in [4].

Elitism The best member of the population may fail to reproduce offspring in the next generation. Examples of fixing this potential source of loss are always keeping the best individual found so far (*elitist strategy*), or conversely, systematically replacing only the worst members of the population with newly generated individuals (*steady-state reproduction*, see Section 1.4.2.2). As shown by Rudolph [31] such a strategy is necessary to ensure global convergence of an evolutionary algorithm. The effect is to shift the balance towards more exploitation and less exploration. However, for some classes of functions this can result in local hill-climbing behavior.

```
            ⋮
#include <EALib/Population.h>
            ⋮
const unsigned PopSize     = 20;
const unsigned NumElitists = 1;

Population parents   ( PopSize, ··· );
Population offsprings( PopSize, ··· );
            ⋮
parents.selectProportional( offsprings, NumElitists );
            ⋮
```

Example 1.6: Proportional selection with elitists.

1.4.1.2 Deterministic Selection

The selection operators described in this section are mainly used in evolutionary strategies and are completely deterministic. Schwefel [34] introduced the notation of (μ, λ)– and $(\mu+\lambda)$–selection.

(μ,λ)-Selection In (μ, λ)–selection the μ parents for the next generation are only taken from the λ offsprings. The previous parent generation is completely discarded, that is the lifetime of each individual is restricted to one generation. The capability of (μ, λ)–selection to forget good solutions in principle allows for leaving local optima and is therefore advantageous in the case of multimodal fitness landscapes or changing environments [2]. The usage of the implemented operators is outlined in Example 1.7.

(μ+λ)-Selection This operator selects the μ best individuals out of the union of parents and offsprings to form the next parent generation. In this respect, $(\mu + \lambda)$–selection can be viewed as the elitist version of (μ, λ)–selection. Since the survival of the best individuals is

guaranteed, the course of evolution is monotonously in that good individuals can only be replaced by better ones. This may be dangerous if the population gets stuck in a poor local optimum.

```
            ⋮
#include <EALib/Population.h>
            ⋮
const unsigned Mu     = 15;
const unsigned Lambda = 100;

Population parents   ( Mu,     ··· );
Population offsprings( Lambda, ··· );
            ⋮
parents.selectMuLambda( offsprings );
            ⋮
```

(a) (μ, λ) - selection.

```
            ⋮
Population parents   ( Mu,     ··· );
Population offsprings( Lambda, ··· );
            ⋮
parents.selectMuLambda( offsprings, Mu );
            ⋮
```

(b) $(\mu + \lambda)$ - selection, the number of elitists is set to *Mu*.

Example 1.7: (μ, λ) and $(\mu + \lambda)$ – selection.

1.4.1.3 Stochastic Selection

A number of stochastic selection operators exist, which base the chance of selection of particular individuals on their associated fitness. A comparative analysis of selection schemes has been carried out in [14]. Depending on what we are looking for, the minimum or maximum fitness, and on the range of the fitness values, the fitness has to be normalized. Normalized fitness values are always positive and total to 1.0 across the entire parent population. In the case of extinctive selection this might be also a sub-population. In this respect the

fitness values can be directly interpreted as selection probabilities. A variety of normalization methods are provided. They rely either on the fitness value itself or on the rank of the respective individual in the population. The rank–based normalization is summarized in special selection operators, whereas fitness-based normalization is performed by special scaling procedures which have to be called beforehand.

```
for i := μ_min to μ_max do
        j := λ_min;
        s := 0;
        x := χ ∈ [0, 1[;
        while s < x and j ≤ λ_max do
                s := s + p_s(a_j(t));
                j := j + 1;
        od
        a_i(t + 1) := a_{j−1}(t);
od
```

Algorithm 1.3: Roulette wheel selection; notation see Table 1.4.1.3.

```
i := μ_min;
s := 0;
x := χ ∈ [0, 1[;
for j := λ_min to λ_max do
        s := s + η(a_j(t));
        while s > x and i ≤ μ_max do
                a_i(t + 1) := a_j(t);
                i := i + 1;
                x := x + 1;
        od
od
```

Algorithm 1.4: Stochastic universal sampling according to [6]; notation see Table 1.4.1.3..

Roulette Wheel Various methods have been suggested for sampling the selection probability.

(a) *Roulette Wheel Selection* [22, 13]. The simplest implementation is to simulate the spin of a weighted roulette wheel. If the localization of the chosen slot is performed via linear search $O(n)$, roulette wheel selection requires $O(n^2)$ steps because in a generation n

spins are necessary to fill the population. If a binary search is used to locate the correct slot the overall complexity reduces to $O(n \log n)$. The standard implementation of roulette wheel selection is outlined in Algorithm 1.3.

(b) *Stochastic Remainder Selection* [8]. In this method the integer portions of the expected number of copies are assigned deterministically. The remainders are selected via the standard roulette wheel. Because $O(n)$ of the individuals are likely to have fractional parts to their reproduction rate, the complexity is $O(n^2)$.

(c) *Stochastic Universal Selection* [6, 16]. Like in (a) the slots of a weighted roulette wheel are sized according to the selection probabilities. Equally space markers are placed along the outside of the wheel and the wheel is spinned only once. The number of markers that fall in each slot determine the number of copies the corresponding individual receives. The complexity of this algorithm is $O(n)$ because only a single pass is needed. An outline of the standard implementation is shown in Algorithm 1.4.

The selection operators presented in the following are completely stochastic in that each individual is assigned a selection probability which determines the chance of the respective individual to be reproduced in the next generation. This holds also for *steady-state* reproduction schemes (cf. Section 1.4.2.2). Before the different operators and the calculations of the respective selection probabilies are presented, we introduce the following notations:

μ	size of the parent population
λ	size of the offspring population from which individuals are selected
μ_{min}, μ_{max}	indices of the subpopulation which is replaced by selected individuals.
$\lambda_{min}, \lambda_{max}$	indices of the subpopulation from which individuals are selected. This is used to model extinctiveness.
a_i	individuals in the offspring population at generation t
$p_s(a_i)$	probability that the ith individual is selected
$\eta(a_i)$	expected number of copies of a_i in the the next generation (reproduction rate of a_i)
$\tilde{\mu}$	$= \mu_{max} - \mu_{min} + 1$
$\tilde{\lambda}$	$= \lambda_{max} - \lambda_{min} + 1$

The selection probabilities have to sum up to 1 across the entire population (λ_{min} – λ_{max}):

$$\sum_{j=\lambda_{min}}^{\lambda_{max}} p_s(a_j) \;=\; 1 \tag{1.1}$$

$\eta(a_i)$ denotes the reproduction rate of individual a_i, that is the expected number of copies of a_i in the next generation:

$$\eta(a_i) \;=\; \begin{cases} p_s(a_i) \cdot (\mu_{max} - \mu_{min} + 1) & , \; \lambda_{min} \leq i \leq \lambda_{max} \\ 0 & , \; i < \lambda_{min} \vee i > \lambda_{max} \end{cases} \tag{1.2}$$

If some of the individuals $\{ a_j, ..., a_{\lambda_{eli}} \}$ with $\{ a_j, ..., a_{\lambda_{eli}} \} \subset \{ a_{\lambda_{min}}, ..., a_{\lambda_{max}} \}$ have been already selected as elitists, we have to correct their selection probabilities. This ensures that the reproduction rate of elitists never exceeds $\max(1, \eta(a_i))$.

$\tilde{p}_s(a_i)$ denotes the corrected selection probability:

$$\Delta p = \frac{1}{\mu_{max} - \mu_{min} + 1} \tag{1.3}$$

$$\Delta s = \sum_{a_i \in \{ a_j, ..., a_{\lambda_{eli}} \}} \min(\Delta p,\, p_s(a_i)) \tag{1.4}$$

$$\tilde{p}_s(a_i) = \begin{cases} (p_s(a_i) - \min(\Delta p,\, p_s(a_i))) \,/\, (1 - \Delta s) & ,\ a_i \in \{ a_j, ..., a_{\lambda_{eli}} \} \\ p_s(a_i) \,/\, (1 - \Delta s) & ,\ a_i \notin \{ a_j, ..., a_{\lambda_{eli}} \} \end{cases} \tag{1.5}$$

Proportional Selection The name proportional selection describes a group of selection schemes that choose individuals for reproduction according to their objective function values, e. g. the probabilty of selection p_s of an individual is proportional to the value:

$$p_s(a_i) = \begin{cases} \Phi(a_i) \,/\, \sum_{j=\lambda_{min}}^{\lambda_{max}} \Phi(a_j) & ,\ \lambda_{min} \le i \le \lambda_{max} \\ 0 & ,\ i < \lambda_{min} \vee i > \lambda_{max} \end{cases} \tag{1.6}$$

Scaling Without scaling, proportional selection is not effective in keeping a steady pressure between competing individuals in the current population [14]. Furthermore, the fitness values have to be positive. If the scaled fitness values total to 1 they are often called normalized fitnesses. A variety of different scaling methods have been proposed. These, mainly, divide into two groups: fitness-based scaling and rank-based scaling. For a detailed description refer to [16, 2]. An example for linear dynamic scaling is outlined in Example 1.8.

Ranking Selection Ranking selection schemes assign each individual a selection probability according to a non–increasing assignment function which is solely based upon the relative rank i of an individual (individuals are assumed to be in descending order with respect to their fitness values). The absolute fitness value is completely ignored, scaling is no longer necessary[1]. Once the selection probabilities are determined, roulette wheel selection or stochastic universal selection is performed. Since sorting the population and sampling of the selection probabilities, as well, can be performed in $O(n \log n)$ ranking selection has the time complexity $O(n \log n)$.

Linear Ranking The most common form of an assignment function is linear [5, 16]:

[1] In principle, this selection scheme can be seen as a combination of rank-based scaling and proportional selection

```
            ⋮
#include <EALib/Population.h>
            ⋮
const unsigned PopSize     = 20;
const unsigned NumElitists = 1;
const unsigned WindowSize  = 5;

//
//  defined scaling window
//
vector< double > window( WindowSize );

//
//  define populations
//
Population parents   ( PopSize, ··· );
Population offsprings( PopSize, ··· );
            ⋮
for( t = 0; t < MaxGeneration; ++t ) {
    //
    //  generate new offsprings (recombination, mutation)
    //
            ⋮
    //
    //  evaluate fitness of offsprings
    //
            ⋮
    //
    //  scale fitness values and use proportional selection
    //
    offsprings.linearDynamicScaling( window, t );
    parents.selectProportional( offsprings, NumElitists );
            ⋮
}
            ⋮
```

Example 1.8: Fitness values are scaled according to the method described in [15].

$$p_s(a_i) \;=\; \begin{cases} \left(\eta_{max} - \dfrac{i-\lambda_{min}}{\lambda_{max}-\lambda_{min}}\,(\eta_{max}-\eta_{min})\right)\dfrac{1}{\lambda_{max}-\lambda_{min}+1} & ,\; \lambda_{min} \le i \le \lambda_{max} \\ 0 & ,\; i < \lambda_{min} \vee i > \lambda_{max} \end{cases} \tag{1.7}$$

```
                ⋮
//
//  define the reproduction rate of the best individual
//
const double EtaMax = 1.1;

Population parents   ( ··· );
Population offsprings( ··· );
                ⋮
parents.selectLinearRanking( offsprings, EtaMax );
                ⋮
```

Example 1.9: Definition of the reproduction rate of the best individual.

Whitley's Linear Ranking With respect to the selection probabilies this scheme is identical to linear ranking (see above), but rather than assigning each individual a particular selection probability it returns directly the index of the individual to be selected [38]. χ denotes a uniformly distributed random number on the interval $[0, 1[$.

$$i(\chi, a) \;=\; \lambda_{min} + \frac{\lambda_{max} - \lambda_{min} + 1}{2\,(a-1)} \cdot \left(a - \sqrt{a^2 - 4\,(a-1)\,\chi}\right) \tag{1.8}$$

The parameter a corresponds to η_{max} in linear ranking. For this selection operator the correction of the selection probabilities of elitists is not defined.

Uniform Ranking Uniform ranking corresponds to linear ranking with $\eta_{max} = 1$.

$$p_s(a_i) \;=\; \begin{cases} \dfrac{1}{\lambda_{max} - \lambda_{min} + 1} & ,\; \lambda_{min} \le i \le \lambda_{max} \\ 0 & ,\; i < \lambda_{min} \vee i > \lambda_{max} \end{cases} \tag{1.9}$$

For given μ and λ it can be viewed as the stochastic version of (μ, λ)–selection.

Tournament Selection In a *q-tournament* the best individual out of randomly sampled group of q individuals is selected [14]. This process is repeated as often as necessary to fill the population for the next generation. For the generational reproduction scheme this is

done by a single call of the selection operator as outlined in Example 1.10. Each competition in the tournament requires the random selection of a constant number of individuals from the population. The comparison among those individuals can be performed in $O(1)$, and n competitions are necessary to fill the population. This results in a time complexity of $O(n)$.

```
            ⋮
//
//  define the tournament size
//
const unsigned Q = 2;

Population parents   ( ··· );
Population offsprings( ··· );
            ⋮
parents.selectTournament( offsprings, Q );
            ⋮
```

Example 1.10: Definition of the tournament size.

EP-Style Tournament Selection In EP-style tournament selection [12], every individual in the parent and offspring population is compared with q randomly selected opponents out of the union of parents and offsprings. For each comparison in which the individuals's fitness is better or equal, it receives a win. Then the μ best individuals out of the union of parents and offsprings form the next parent generatiton. In EP-style tournament selection an individual is considered to be better than another if it recieved more wins in the last tournament. If two individuals recieved the same number of wins, the one with the better fitness value is considered to be better. EP-style tournament selection is an elitist, preservative strategy.

1.4.2 Replacement

A number of candidates (the offsprings) are generated from the current population, e. g. by recombination and mutation, and may replace some or all of the parents top produce the next generation.

1.4.2.1 Generational Replacement

In the generational replacement scheme the entire population is replaced at once. The generational replacement technique has some potential drawbacks. Even with an elitist strategy many of the best individuals found may not reproduce at all or their chromosomes may be lost due to crossover and . But generational replacement is fairly robust against noisy objective functions and changing environments since each individual is re-evaluated whenenver it turns up in the population.

1.4.2.2 Steady-State Replacement

In the steady-state replacement scheme only a few members are replaced at a time [38, 36]. Steady-state replacement has one parameter—the number of new individuals to create (the number of old individuals to replace, respectively). Note that generational replacement is a special case of steady-state replacement where the number of replaced individuals equals the population size. For a comparison of the two replacement schemes see [37].

The condition can be imposed that only one copy of any individual exists in the population at any time so that the population never fully converges. This variant is called *steady-state without duplicates*. It involves some computational overhead, in that it requires a large number of equality tests whenever an offspring is created, but applying genetic algorithms to *real* problems most of the time is spent in evaluation. So this additional computation time is negligible with regard to the total computing time.

It should be noted that steady-state replacement may not work well when the objective function is noisy or the environment is changing since good individuals won't be deleted from the population even if they are bad in the changed environment. For further discussion see [10], chapter 2.

1.4.3 Recombination

After the individuals have been selected for reproduction they mate by recombining their respective chromosomes. Depending on the representation of the chromosome different recombination operators can be applied. The most universal operator is the *crossing over* which is, in principle, applicable for all chromosomes which can be described by a sequence of values. This is the standard recombination operator in GAs. The recombination of continuous values, as needed in ESs, is introduced in Section 1.4.3.2.

1.4.3.1 Crossover of Strings

In most GAs, individuals are represented by fixed-length strings of symbols of a finite alphabet. Recombination is implemented by means of crossing-over. The crossover operator acts on pairs of individuals (parents) and creates new individuals (offsprings) by exchanging segments from the parents' strings. The number and the position of these segments are determinded by the crossover points. Usually these crossover points are sampled randomly. As a constraint the granularity of crossover can be controlled so that crossover points are not selected at any point within the string. This may be useful if chunks of consecutive symbols encode single items which should not be disrupted by crossover. A comparison and analysis of the most popular methods to choose crossover points is presented in [35]. The probability of the application of the crossover operator is controlled by the crossover rate.

m-Point Crossover The disruptiveness is controlled by the number of crossover points m. Figure 1.2 shows examples of one point and two point crossover.

Uniform Crossover Syswerda [36] defines a family of *uniform* crossover operators which produce offspring by randomly selecting at each loci the allele of one of the parents. This

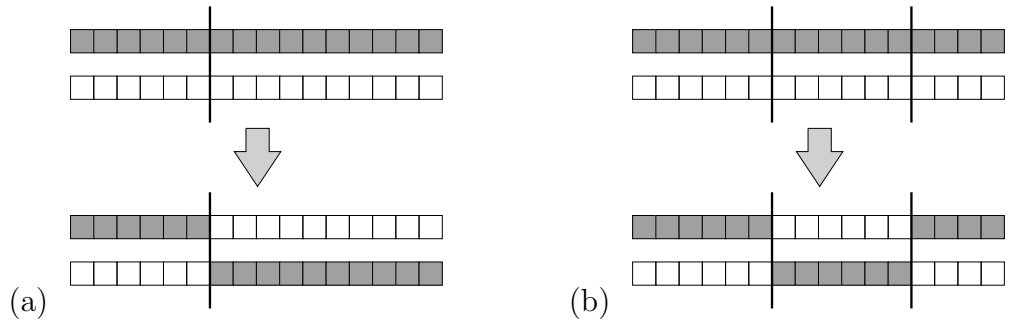

Figure 1.2: (a) One point crossover. (b) Two point crossover.

corresponds on average to a $(L/2)$-point crossover on strings of length L. Uniform crossover is an adaptation of Mendel's chance model to haploid organisms [1].

Multi-Parent Crossover A multi-parent operator uses two or more parents to generate offsprings. Gene scanning, as discussed in [11], generalizes uniform-crossover to take more than two parents into account. For each position in the offspring a parent is chosen to contribute its allele. There are different methods to choose a parent.

- Uniform scanning. For each allele in the offspring a parent is chosen randomly. Each parent has an equal chance of contributing an allele to be inherited by the offspring.
- Occurence-based scanning. The allele which occurs most often is chosen for inheritance.
- Fitness-based scanning. The probability of a parent to be selected is proportional to its fitness.

1.4.3.2 Recombination of Continuous Values

In the field of ESs a variety of recombination operators have been proposed, including sexual as well as panmictic operators. In the following the most frequently employed operators are shortly introduced.

Discrete Recombination Discrete recombination means the same as uniform crossover, that is for each allele one of the (two) parents is randomly selected for inheritance.

Intermediate Recombination In this form of recombination the new allele is a weighted sum of the alleles of the corresponding parents. In the simplest form this sum is the arithmetic mean of all parent alleles.

Panmictic Recombination Panmictic recombination involves more than two parents. Combined with discrete recombination this means that for each allele in the offspring individual a parent is sampled anew. In the case of intermediate recombination each parent may contribute to the weighted sum of alleles.

1.4.4 Mutation

After selection and recombination the reproduced individuals are subject to mutation. The main differences between the mutation operators used in GAs and ESs are due to the different representation of the genome. In GAs the genome is encoded into symbol strings and therefore discrete mutations apply, whereas in ESs floating point numbers are used combined with continuous mutations (see Section 1.4.4.2).

1.4.4.1 Discrete Mutations

The standard mutation operator for GAs was first proposed by Holland [21]. Each position of the genome is given a chance p_m (the mutation rate) of undergoing mutation. In the case of mutation a random value is chosen from the set of allowed symbols for the selected allele. Using binary strings a simpler mechanism can be employed. For each position to be mutated the corresponding bit is simply flipped.

Adaptive/Variable Mutation Rates Adopting the idea of self-adaptation of the strategy parameters from ES, variable mutation rates can be defined. This has been proposed by Bäck, results can be found in [4].

1.4.4.2 Continuous Mutations

ESs rely heavily on the mutation operator, which adds a normally distributed random number to each allele. The step size of the mutation is controlled by the standard deviation of the normal distribution.

Correlated Mutations In order to allow for more flexible distributions in the mutation operator correlations may be considered. Schwefel [34] has proposed a correlated mutation scheme where the correlations were implicitly defined by a rotation matrix (see also [30]).

1.4.4.3 Inversion

Holland [21] describes an inversion operator which works on a single chromosome. It inverts the order of the elements between two randomly chosen points on the chromosome. While inversion was inspired by a biological mechanism it has not in general been found to be useful in genetic algorithms. In combination with a transcription operator which works on chromosomes that are several times longer, inversion proves to be useful in some applications [39].

1.4.5 Self-adaptation in Evolution Strategies

Evolution strategies and many other evolutionary algorithms for real-valued optimization usually rely on Gaussian random mutations. That is, candidate solutions are altered by adding random vectors drawn according to zero-mean normal distributions. Adaptation of the covariance matrices of these random variations during optimization allows for learning

and employing an appropriate metric for the search process. It is well known that an automatic adaptation of the mutation distribution drastically improves the search performance on non-separable or badly scaled objective functions [29, 33, 20, 7, 23].

The original approaches to the adaptation of the covariance matrix add parameters describing the matrix to the chromosome of each individual and rely on a "second-order" selection process for their choice. This is termed self-adaptation of strategy parameters in ES. In the following, we will give a short overview over the different methods, which are implemented in the library.

The "derandomized" adaptation put forward by Ostermeier [26, 27] aims at a more direct adaptation of the strategy parameters. In the standard evolution strategy the adaptation has to overcome the stochastic fluctuations which are introduced by the fact that even distributions with small variances can produce large values and vice versa. This source of "noise" in the adaptation process is circumvented by using the actual step-length, that is, the realization of the random number z for the adaptation. In addition, to the derandomization the idea of an "evolution path" is inherent in all of these approaches. The principle is very similar to a momentum term in standard gradient descent approaches. The adaptation does not just rely on the step which is chosen in the current generation but also on selected steps in previous generation.

1.4.5.1 Standard adaptation

In the standard evolution strategy, the mutation of the objective variables $\mathbf{x}$ is carried out by adding a $N(0, \sigma_i^2)$ distributed random number z_i to each component of x_i. The "step-sizes" σ_i are also subject to mutations (log-normal distributed) both for each component separately (parameterized by τ) and overall (parameterized by τ'). Thus, the individual consists of both the objective and the step-size vector.

$$\begin{aligned} \sigma_i^{(t)} &= \sigma_i^{(t-1)} \exp(\tau' z) \ \exp(\tau z_i) && (1.10)\\ \mathbf{x}^{(t)} &= \mathbf{x}^{(t-1)} + \tilde{\mathbf{z}} && (1.11)\\ z_i, z &\sim N(0,1) && (1.12)\\ \tilde{\mathbf{z}} &\sim N\left(\mathbf{0}, \left(\tilde{\sigma}^{(t)}\right)^2\right). && (1.13) \end{aligned}$$

This is the standard evolution strategy according to [3, 33]. The version put forward by [29] is different, particularly with respect to the changes of the "step-size":

$$\begin{aligned} \sigma_i^{(t)} &= \sigma_i^{(t-1)} \xi, && (1.14)\\ \mathbf{x}^{(t)} &= \mathbf{x}^{(t-1)} + \sigma \cdot \mathbf{z} && (1.15)\\ z_i &\sim N(0, 1/n) && (1.16) \end{aligned}$$

ξ has the value 1.5 with probability ξ_{prob} and the value 1/1.5 with probability $(1 - \xi_{prob})$. In the standard method in [29], $\xi_{prob} = 0.5$. n is the dimension of the objective vector. The variance $1/n$ guarantees that the length $||z||$ is one on average for large n.

1.4.5.2 Rotation matrix adaptation

Instead of adapting the complete correlation matrix $\mathbf{C}$ of the n-dimensional normal distribution (eq. 1.17), it is sufficient to rotate the diagonal matrix $\mathbf{C}' = diag(\sigma_1, \ldots, \sigma_n)$ using

Parameter	Standard Value	Comment
τ'	$1/\sqrt{2n}$	heuristic value
τ	$1/\sqrt{2\sqrt{n}}$	heuristic value
β	0.0873	heuristic value
toggle	FALSE	enforce lower bound
ϵ		lower bound for σ

Table 1.1: Parameters for the rotation angle adaptation and their standard values; n is the dimension of the objective vector $\mathbf{x}$.

$n(n-1)/2$ rotation angles.

$$p(\mathbf{z}) = \frac{1}{(2\pi)^{n/2}\det\mathbf{C}}\exp\left(-\frac{1}{2}\mathbf{z}^T\,\mathbf{C}^{-1}\,\mathbf{z}\right). \tag{1.17}$$

The rotations can be written as follows using the rotation matrices $\mathbf{R}(\alpha_{ij})$:

$$\mathbf{R}(\alpha_{ij}) = \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & \cdots & \cdots & 0 \\ \cdots & \cos\alpha_{ij} & \cdots & -\sin\alpha_{ij} & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \sin\alpha_{ij} & \cdots & \cos\alpha_{ij} & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & \cdots & 0 & 1 \end{pmatrix} \tag{1.18}$$

$$\tilde{\sigma} = \left(\prod_{i=1}^{n-1}\prod_{j=i+1}^{n}\mathbf{R}(\alpha_{ij})\right)\cdot\sigma \tag{1.19}$$

The trigonometric functions occur in $\mathbf{R}(\alpha_{ij})$ at the positions $(row, column) = (i,i),(i,j),(j,i),(j,j)$. Mutation of the n step sizes σ and the mutation angles is carried out like in the standard algorithm

$$\sigma_i^{(t)} = \sigma_i^{(t-1)}\exp(\tau' z)\ \exp(\tau\, z_i) \tag{1.20}$$

$$\alpha_{ij}^{(t)} = \alpha_{ij}^{(t-1)} + \beta\, z_{ij}^{\alpha} \tag{1.21}$$

$$\mathbf{x}^{(t)} = \mathbf{x}^{(t-1)} + \tilde{\mathbf{z}} \tag{1.22}$$

$$z_i, z, z_{ij}^{\alpha} \sim N(0,1) \tag{1.23}$$

$$\tilde{\mathbf{z}} \sim N(\mathbf{0}, \tilde{\sigma}^{(t)}). \tag{1.24}$$

The values of α_{ij} are confined to the interval $[-\pi,\pi]$ and a lower bound can be defined for the σ_i (in the function this is determined by the values of *toggle* and ϵ). The standard setting of the parameters is shown in Table 1.1, two versions of the operator **mutateRotate** exist, one which includes the standard values and one where all variables have to be defined explicitly.

1.4.5.3 The $(\mu_{\mathbf{w}}, \lambda)$-CMA Evolution Strategy

Basic Principles In the CMA-ES [18, 17] a parent population of μ candidate solutions are maintained, from which $\lambda > \mu$ new candidate solutions are generated in each iteration. The performances of these offspring solutions are determined and the μ best form the parent population in the next iteration. In each iteration, the kth offspring $\mathbf{x}_k \in \mathbb{R}^n$ is generated by multi-variate *Gaussian mutation* and *weighted global intermediate recombination*, i.e.,

$$\mathbf{x}_k = \langle \mathbf{x}_{\text{parents}} \rangle_{\mathbf{w}} + \sigma \mathbf{z}_k,$$

where $\mathbf{z}_k \sim \mathrm{N}(\mathbf{0}, \mathbf{C})$ and

$$\langle \mathbf{x}_{\text{parents}} \rangle_{\mathbf{w}} = \sum_{i=1}^{\mu} w_i \mathbf{x}_{i\text{th-best-parent}}$$

(a common chice is $\mathbf{w}_i \propto \ln(\mu + 1) - \ln(i)$, $\|\mathbf{w}\|_1 = 1$). The CMA-ES is a variable metric algorithm adapting both the n-dimensional covariance matrix $\mathbf{C}$ of the normal mutation distribution as well as the *global step size* $\sigma \in \mathbb{R}^+$. In the basic algorithm, a low-pass filtered *evolution path* $\mathbf{p}$ of successful directions (i.e., selected steps) is stored,

$$\mathbf{p} \leftarrow \eta_1 \, \mathbf{p} + \eta_2 \, \left(\langle \mathbf{x}_{\text{new parents}} \rangle - \langle \mathbf{x}_{\text{old parents}} \rangle \right),$$

and $\mathbf{C}$ is changed to make steps in the direction $\mathbf{p}$ more likely: $\mathbf{C} \leftarrow \eta_3 \, \mathbf{C} + \eta_4 \, \mathbf{p}\mathbf{p}^T$ (this rank-one update of $\mathbf{C}$ is augmented by a rank-μ update, see [18]). The variables $\eta_1, \ldots, \eta_4$ denote fixed learning rates and normalization constants set to default values, see below. The global step size σ is adapted on a faster timescale. It is increased if the selected steps are larger and/or more correlated than expected and decreased if they are smaller and/or more anticorrelated than expected. The highly efficient use of information and the fast adaptation of σ and $\mathbf{C}$ makes the CMA-ES one of the best direct search algorithms for real-valued optimization.

Details In the following, we present the details of the covariance matrix adaptation ES with weighted recombination and rank-μ-update $(\mu_{\mathbf{w}}, \lambda)$-CMA-ES) as described in [18, 17], which is based on [19, 20].

Each individual represents an n-dimensional real-valued object variable vector. These variables are altered by two variation operators, intermediate recombination and additive Gaussian mutation. The former corresponds to computing the (possibly weighted) center of mass of the μ individuals in the parent population. Mutation is realized by adding a normally distributed random vector with zero mean. The complete covariance matrix of the Gaussian mutation distribution is adapted during evolution to improve the search strategy.

The object parameters $\mathbf{x}_k^{(t+1)}$ of offspring $k = 1, \ldots, \lambda$ created in generation $g + 1$ are given by

$$\mathbf{x}_k^{(t+1)} = \langle \mathbf{x} \rangle_{\mathbf{w}}^{(t)} + \sigma^{(t)} \mathbf{B}^{(t)} \mathbf{D}^{(t)} \mathbf{z}_k^{(t)} \quad , \tag{1.25}$$

where the $\mathbf{z}_k^{(t)} \sim N(\mathbf{0}, \mathbf{I})$ are independent realizations of an n-dimensional normally distributed random vector with zero mean and covariance matrix equal to the identity matrix $\mathbf{I}$ and

$$\langle \mathbf{x} \rangle_{\mathbf{w}}^{(t)} = \sum_{i=1}^{\mu} w_i \mathbf{x}_{i:\lambda}^{(t)} \tag{1.26}$$

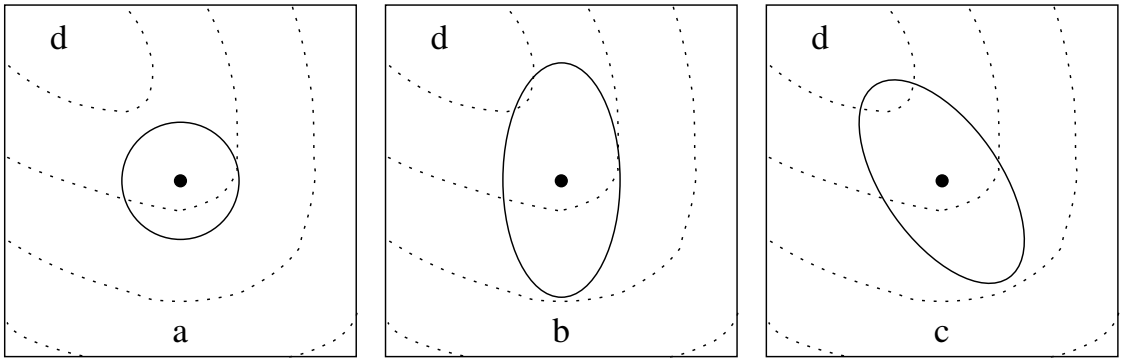

Figure 1.3: The dashed lines schematically visualize an error / fitness surface (landscape) for $\subset \mathbb{R}^2$, where each line represents points of equal fitness and the $\times$ symbol marks the optimum. The dot corresponds to the center of mass of the parent population and the solid lines indicate the mutation (hyper-) ellipsoids (i.e., surfaces of equal probability density to place an offspring) of the random vectors after the different transformations. Evolution strategies that adapt only one global step size can only produce mutation ellipsoids as shown in the left plot. Algorithms that adapt n different step sizes, one for each object variable, can produce mutation ellipsoids scaled along the coordinate axes like the one shown in the center plot. Only if the complete covariance matrix is adapted, arbitrary normal distributions can be realized as shown in the right picture.

is the weighted mean of the selected individuals with

$$\sum_{i=1}^{\mu} w_i = 1 \tag{1.27}$$

and $w_i > 0$ for $i = 1, \ldots, \mu$. The index $i : \lambda$ denotes the ith best individual. The covariance matrix $\mathbf{C}^{(t)}$ of the random vectors

$$\mathbf{B}^{(t)}\mathbf{D}^{(t)}\mathbf{z}_k^{(t)} \sim N(\mathbf{0}, \mathbf{C}^{(t)}) \tag{1.28}$$

is a symmetric positive $n \times n$ matrix with

$$\mathbf{C}^{(t)} = \mathbf{B}^{(t)}\mathbf{D}^{(t)} \left(\mathbf{B}^{(t)}\mathbf{D}^{(t)}\right)^T \quad . \tag{1.29}$$

The columns of the orthogonal $n \times n$ matrix $\mathbf{B}^{(t)}$ are the normalized eigenvectors of $\mathbf{C}^{(t)}$ and $\mathbf{D}^{(t)}$ is a $n \times n$ diagonal matrix with the square roots of the corresponding eigenvalues. Figure 1.3 schematically shows the transformations of $\mathbf{z}_k^{(t)}$ by $\mathbf{B}^{(t)}$ and $\mathbf{D}^{(t)}$.

The strategy parameters, both the matrix $\mathbf{C}^{(t)}$ and the so called global step-size $\sigma^{(t)}$, are updated online using the covariance matrix adaptation (CMA) method. The key idea of the CMA is to alter the mutation distribution in a deterministic way such that the probability to reproduce steps in the search space that have led to the current population is increased. This enables the algorithm to detect correlations between object variables and to become invariant under orthogonal transformations of the search space (apart from the initialization). In order to use the information from previous generations efficiently, the search path of the population over a number of past generations is taken into account.

In the CMA-ES, rank-based (μ, λ)-selection is used for environmental selection. That is, the μ best of the λ offspring form the next parent population. After selection, the strategy parameters are updated:

$$\mathbf{p}^{(t+1)} = (1 - c_c) \cdot \mathbf{p}_c^{(t)} + \sqrt{c_c(2 - c_c)} \frac{\sqrt{\mu_{\text{eff}}}}{\sigma^{(t)}} \left(\langle \mathbf{x} \rangle^{(t+1)} - \langle \mathbf{x} \rangle^{(t)} \right) \quad , \tag{1.30}$$

$$\begin{aligned} \mathbf{C}^{(t+1)} = (1 - c_{\text{cov}}) \cdot \mathbf{C}^{(t)} + c_{\text{cov}} \cdot \Bigg(& \frac{1}{\mu_{\text{cov}}} \mathbf{p}^{(t+1)} \left(\mathbf{p}^{(t+1)} \right)^{\mathrm{T}} \\ & + \left(1 - \frac{1}{\mu_{\text{cov}}} \right) \sum_{i=1}^{\mu} \frac{w_i}{{\sigma^{(t)}}^2} \left(\mathbf{x}_{i:\lambda}^{(t+1)} - \langle \mathbf{x} \rangle_{\mathbf{w}}^{(t)} \right) \left(\mathbf{x}_{i:\lambda}^{(t+1)} - \langle \mathbf{x} \rangle_{\mathbf{w}}^{(t)} \right)^T \Bigg) \quad . \end{aligned} \tag{1.31}$$

Herein, $\mathbf{p}^{(t+1)} \in \mathbb{R}^n$ is the evolution path—a weighted sum of the centers of the population over the generations starting from $\mathbf{p}^{(0)} = \mathbf{0}$ (the factor $\sqrt{\mu_{\text{eff}}}$ compensates for the loss of variance due to computing the center of mass). The parameter $c_c \in]0, 1]$ controls the time horizon of the adaptation of $\mathbf{p}$. The constant $\sqrt{c_c(2 - c_c)}$ normalizes the variance of $\mathbf{p}$ (viewed as a random variable) as $1^2 = (1 - c)^2 + (\sqrt{c_c(2 - c_c)})^2$. The index $i{:}\lambda$ is the index of the offspring having the ith best fitness value of all offspring in the current generation. The parameter $c_{\text{cov}} \in [0, 1[$ controls the update of $\mathbf{C}^{(t)}$. The vector $\mathbf{p}$ does not only represent the last (adaptive) step of the parent population, but a time average over all previous adaptive steps. The influence of previous steps decays exponentially, where the decay rate is controlled by c_{cov}.

The first line of the update rule (1.31) shifts $\mathbf{C}^{(t)}$ towards the $n \times n$ matrix $\mathbf{p}^{(t+1)} \left(\mathbf{p}^{(t+1)} \right)^T$, which has rank 1, making mutation steps in the direction of $\mathbf{p}^{(t+1)}$ more likely. The second line is the rank-μ-update, which is particularly useful in case of large populations: the matrix resulting from the sum over all selected offspring has rank $\min(\mu, n)$.

The adaptation of the global step-size parameter σ is done separately on a faster timescale (a single parameter can be estimated based on less samples compared to the complete covariance matrix). We keep track of a second evolution path $\mathbf{p}_\sigma$ without the scaling by $\mathbf{D}$:

$$\begin{aligned} \mathbf{p}_\sigma^{(t+1)} = & (1 - c_\sigma) \cdot \mathbf{p}_\sigma^{(t)} \\ & + \sqrt{c_\sigma(2 - c_\sigma)} \cdot \mathbf{B}^{(t)} {\mathbf{D}^{(t)}}^{-1} {\mathbf{B}^{(t)}}^T \frac{\sqrt{\mu_{\text{eff}}}}{\sigma^{(t)}} \left(\langle \mathbf{x} \rangle^{(t+1)} - \langle \mathbf{x} \rangle^{(t)} \right) \quad , \end{aligned} \tag{1.32}$$

$$\sigma^{(t+1)} = \sigma^{(t)} \cdot \exp \left(\frac{c_\sigma}{d_\sigma} \left(\frac{\| \mathbf{p}_\sigma^{(t+1)} \| - \hat{\chi}_n}{\hat{\chi}_n} \right) \right) \quad , \tag{1.33}$$

where $\hat{\chi}_n$ is the expected length of a n-dimensional, normally distributed random vector with covariance matrix $\mathbf{I}$. It is approximated by $\sqrt{n}(1 - \frac{1}{4n} + \frac{1}{21n^2}$. The damping parameter d_{sigma} decouples the adaptation rate from the strength of the variation. The parameter $c_\sigma \in]0, 1]$ controls the update of $\mathbf{p}_\sigma$.

The evolution path $\mathbf{p}_\sigma$ is the sum of normally distributed random variables starting from $\mathbf{p}_\sigma^{(0)} = \mathbf{0}$. Because of the normalization, its expected length would be $\hat{\chi}_n$ if there were no selection. Hence, the rule (1.33) basically increases the global step-size if the steps leading to selected individuals have been larger and/or more correlated than expected and decreased if they are smaller and/or more anticorrelated than expected.

The standard parameters using the rank-μ-update and superlinear weighted recombination are

$$\lambda = 4 + \lfloor 3 \ln n \rfloor \ , \tag{1.34}$$
$$\mu = \lfloor \lambda/2 \rfloor \ , \tag{1.35}$$
$$w_i \propto \ln(\mu + 1) - \ln i \ , \tag{1.36}$$
$$c_\sigma = \frac{\mu_{\text{eff}} + 2}{n + \mu_{\text{eff}} + 3} \ , \tag{1.37}$$
$$d_\sigma = 1 + 2 \max\left(0, \sqrt{\frac{\mu_{\text{eff}} - 1}{n+1}}\right) + c_\sigma \ , \tag{1.38}$$
$$c_c = \frac{4}{4+n} \ , \tag{1.39}$$
$$\mu_{\text{cov}} = \mu_{\text{eff}} \ , \tag{1.40}$$
$$c_{\text{cov}} = \frac{1}{\mu_{\text{cov}}} \frac{2}{(n+\sqrt{2})^2} + \left(1 - \frac{1}{\mu_{\text{cov}}}\right) \min\left(1, \frac{2\mu_{\text{eff}} - 1}{(n+2)^2 + \mu_{\text{eff}}}\right) \ . \tag{1.41}$$

If linear weighted recombination is used we have

$$w_i \propto \mu + 1 - i \ . \tag{1.42}$$

The weights w_i are normalized according to (1.27). If no weighted recombination is used the standard parameters are

$$w_i = 1/\mu \ , \tag{1.43}$$
$$\mu = \lfloor \lambda/4 \rfloor \ . \tag{1.44}$$

If no rank-μ-update is used, which can be reasonable for small population sizes,

$$\mu_{\text{cov}} = 1 \tag{1.45}$$

and the other parameters are set accordingly.

A CMA object is initialized in the following way:

```
cma.init(n,
         variances,
         σ(0),
         population,
         recombinationType,  // default:  CMA::superlinear
         updateType,         // default:  CMA::rankmu
         initType);          // don't change default value
```

The `double` vector `variances` contains the diagonal elements of $\mathbf{C}^{(0)}$. The recombination type can be `CMA::equal`, `lCMA::inear`, or `CMA::superlinear`. The type of update can be

chosen from `CMA::rankone` and `CMA::rankmu`. The CMA parameters are set based on on the recombination type and the update type as described above. the membership functions `suggestMu(`n`)` and `suggestLambda(`n`, recombinationType)` return recommended population sizes according to the heuristics given above. In addition, the suggested λ is bounded from below below by 5 as recommended by N. Hansen and A. Ostermeier and from above by n.

The initializations of $\mathbf{C}$ and σ allow for incorporation of prior knowledge about the scaling of the search space. In the following, we set $\mathbf{C}^{(1)} = \mathbf{I}$ and choose $\sigma^{(1)}$ dependent on the problem. The example in section 1.5.4 shows how the CMA-ES is used.

1.5 Examples

1.5.1 Canonical Genetic Algorithm

```
#include <EALib/PopulationT.h>

using namespace std;

//===========================================================================
//
// fitness function:  counting ones problem
//
double ones( const vector< bool >& x )
{
    unsigned i;
    double   sum;
    for( sum = 0., i = 0; i < x.size( ); sum += x[ i++ ] );
    return sum;
}

//===========================================================================
//
// main program
//
int main( int argc, char **argv )
{
    //
    // constants
    //
    const unsigned PopSize     = 20;
    const unsigned Dimension   = 200;
    const unsigned Iterations  = 1000;
    const unsigned Interval    = 10;
    const unsigned NElitists   = 1;
    const unsigned Omega       = 5;
    const unsigned CrossPoints = 2;
    const double   CrossProb   = 0.6;
    const double   FlipProb    = 1.  / Dimension;

    unsigned i, t;

    //
    // initialize random number generator
    //
```

Program 1.1

```
Rng::seed( argc > 1 ?  atoi( argv[ 1 ] ) :  1234 );

//
// define populations
//
PopulationT<bool> parents   ( PopSize, ChromosomeT< bool >( Dimension ) );
PopulationT<bool> offsprings( PopSize, ChromosomeT< bool >( Dimension ) );

//
// scaling window
//
vector< double > window( Omega );

//
// maximization task
//
parents   .setMaximize( );
offsprings.setMaximize( );

//
// initialize all chromosomes of parent population
//
for( i = 0; i < parents.size( ); ++i )
        parents[ i ][ 0 ].initialize( );

//
// evaluate parents (only needed for elitist strategy)
//
if( NElitists > 0 )
    for( i = 0; i < parents.size( ); ++i )
        parents[ i ].setFitness( ones(parents[ i ][ 0 ] ) );

//
// iterate
//
for( t = 0; t < Iterations; ++t ) {
    //
    // recombine by crossing over two parents
    //
    offsprings = parents;
    for( i = 0; i < offsprings.size( )-1; i += 2 )
        if( Rng::coinToss( CrossProb ) )
            offsprings[ i ][ 0 ]
                .crossover( offsprings[ i+1 ][ 0 ], CrossPoints );
    //
```

Program 1.1

```
        // mutate by flipping bits
        //
        for( i = 0; i < offsprings.size( ); ++i )
            offsprings[ i ][ 0 ].flip( FlipProb );
        //
        // evaluate objective function
        //
        for( i = 0; i < offsprings.size( ); ++i )
            offsprings[ i ].setFitness( ones(offsprings[ i ][ 0 ] ) );
        //
        // scale fitness values and use proportional selection
        //
        offsprings.linearDynamicScaling( window, t );
        parents.selectProportional( offsprings, NElitists );

        //
        // print out best value found so far
        //
        if( t % Interval == 0 )
            cout << t << "\tbest value = "
                 << parents.best( ).fitnessValue( ) << endl;
    }
}
```

Program 1.1: Canonical Genetic Algorithm solving counting ones problem.

```
#include <SharkDefs.h>
#include <EALib/PopulationT.h>

using namespace std;

//=============================================================================
//
// fitness function:  sphere model
//
double sphere( const vector< double >& x )
{
    unsigned i;
    double   sum;
    for( sum = 0., i = 0; i < x.size( ); i++ )
        sum += Shark::sqr( x[ i ] );
    return sum;
}

//=============================================================================
//
// main program
//
int main( int argc, char **argv )
{
    //
    // constants
    //
    const unsigned PopSize     = 50;
    const unsigned Dimension   = 20;
    const unsigned NumOfBits   = 10;
    const unsigned ChromLen    = Dimension * NumOfBits;
    const unsigned Iterations  = 2000;
    const unsigned DspInterval = 10;
    const unsigned NElitists   = 1;
    const unsigned Omega       = 5;
    const unsigned CrossPoints = 2;
    const double   CrossProb   = 0.6;
    const double   FlipProb    = 1.  / ChromLen;
    const bool     UseGrayCode = true;
    const Interval RangeOfValues( -3, +5 );

    unsigned i, t;

    //
    // initialize random number generator
```

Program 1.2

```
//
Rng::seed( argc > 1 ?  atoi( argv[ 1 ] ) :  1234 );

//
// define populations
//
PopulationT<bool> parents   ( PopSize, ChromosomeT< bool >( ChromLen ) );
PopulationT<bool> offsprings( PopSize, ChromosomeT< bool >( ChromLen ) );

//
// scaling window
//
vector< double > window( Omega );

//
// temporary chromosome for decoding
//
ChromosomeT< double > dblchrom;

//
// minimization task
//
parents   .setMinimize( );
offsprings.setMinimize( );

//
// initialize all chromosomes of parent population
//
for( i = 0; i < parents.size( ); ++i )
    parents[ i ][ 0 ].initialize( );

//
// evaluate parents (only needed for elitist strategy)
//
if( NElitists > 0 )
    for( i = 0; i < parents.size( ); ++i ) {
        dblchrom.decodeBinary( parents[ i ][ 0 ], RangeOfValues,
                               NumOfBits, UseGrayCode );
        parents[ i ].setFitness( sphere( dblchrom ) );
    }

//
// iterate
//
for( t = 0; t < Iterations; ++t ) {
```

Program 1.2

```
        //
        // recombine by crossing over two parents
        //
        offsprings = parents;
        for( i = 0; i < offsprings.size( )-1; i += 2 )
            if( Rng::coinToss( CrossProb ) )
                offsprings[ i ][ 0 ]
                  .crossover( offsprings[ i+1 ][ 0 ], CrossPoints );

        //
        // mutate by flipping bits
        //
        for( i = 0; i < offsprings.size( ); ++i )
            offsprings[ i ][ 0 ].flip( FlipProb );

        //
        // evaluate objective function
        //
        for( i = 0; i < offsprings.size( ); ++i ) {
            dblchrom.decodeBinary( offsprings[ i ][ 0 ], RangeOfValues,
                                   NumOfBits, UseGrayCode );
            offsprings[ i ].setFitness( sphere( dblchrom ) );
        }

        //
        // scale fitness values and use proportional selection
        //
        offsprings.linearDynamicScaling( window, t );
        parents.selectProportional( offsprings, NElitists );

        //
        // print out best value found so far
        //
        if( t % DspInterval == 0 )
            cout << t << "\tbest value = "
                 << parents.best( ).fitnessValue( ) << endl;
    }
}
```

Program 1.2: Canonical Genetic Algorithm minimizing the sphere model.

1.5.2 Steady-State Genetic Algorithm

```
#include <SharkDefs.h>
#include <EALib/PopulationT.h>

using namespace std;

//=============================================================================
//
// fitness function:  sphere model
//
double sphere( const vector< double >& x )
{
    unsigned i;
    double   sum;
    for( sum = 0., i = 0; i < x.size( ); i++ )
        sum += Shark::sqr( x[ i ] );
    return sum;
}

//=============================================================================
//
// main program
//
int main( int argc, char **argv )
{
    //
    // constants
    //
    const unsigned PopSize     = 50;
    const unsigned Dimension   = 20;
    const unsigned NumOfBits   = 10;
    const unsigned Iterations  = 15000;
    const unsigned DspInterval = 100;
    const unsigned Omega       = 5;
    const unsigned CrossPoints = 2;
    const double   CrossProb   = 0.6;
    const double   FlipProb    = 1.  / ( Dimension * NumOfBits );
    const bool     UseGrayCode = true;
    const Interval RangeOfValues( -3, +5 );

    unsigned i, t;

    //
```

Program 1.3

```
// initialize random number generator
//
Rng::seed( argc > 1 ?  atoi( argv[ 1 ] ) :  1234 );

//
// define populations
//
IndividualT<bool> kid( ChromosomeT< bool >( Dimension * NumOfBits ) );
PopulationT<bool> pop( PopSize, kid );

//
// scaling window
//
vector< double > window( Omega );

//
// temporary chromosome for decoding
//
ChromosomeT< double > dblchrom;

//
// minimization task
//
pop.setMinimize( );

//
// initialize all chromosomes of the population
//
for( i = 0; i < pop.size( ); ++i ) {
    pop[ i ][ 0 ].initialize( );
    dblchrom.decodeBinary( pop[ i ][ 0 ], RangeOfValues,
                           NumOfBits, UseGrayCode );
    pop[ i ].setFitness( sphere( dblchrom ) );
}

//
// iterate
//
for( t = 0; t < Iterations; ++t ) {
    //
    // scale fitness values and use proportional selection
    //
    pop.linearDynamicScaling( window, t );

    //
```

Program 1.3

```
        // recombine by crossing over two parents
        //
        if( Rng::coinToss( CrossProb ) )
            kid[ 0 ].crossover( pop.selectOneIndividual( )[ 0 ],
                                pop.selectOneIndividual( )[ 0 ],
                                CrossPoints );
        else
            kid = pop.selectOneIndividual( );

        //
        // mutate by flipping bits
        //
        kid[ 0 ].flip( FlipProb );

        //
        // evaluate objective function
        //
        dblchrom.decodeBinary( kid[ 0 ], RangeOfValues,
                               NumOfBits, UseGrayCode );
        kid.setFitness( sphere( dblchrom ) );

        //
        // replace the worst individual in the population
        //
        pop.worst( ) = kid;

        //
        // print out best value found so far
        //
        if( t % DspInterval == 0 )
            cout << t << "\tbest value = "
                 << pop.best( ).fitnessValue( ) << endl;
    }
}
```

Program 1.3: Steady–State Genetic Algorithm solving the counting ones problem.

1.5.3 Canonical Evolution Strategy

```
#include <SharkDefs.h>
#include <EALib/PopulationT.h>

using namespace std;

//===========================================================================
//
// fitness function:  Ackley's function
//
double ackley( const vector< double >& x )
{
    const double A = 20.;
    const double B = 0.2;
    const double C = M_2PI;

    unsigned i, n;
    double   a, b;

    for( a = b = 0., i = 0, n = x.size( ); i < n; ++i ) {
        a += x[ i ] * x[ i ];
        b += cos( C * x[ i ] );
    }

    return -A * exp( -B * sqrt( a / n ) ) - exp( b / n ) + A + M_E;
}

//===========================================================================
//
// main program
//
int main( int argc, char **argv )
{
    //
    // constants
    //
    const unsigned Mu           = 15;
    const unsigned Lambda       = 100;
    const unsigned Dimension    = 30;
    const unsigned Iterations   = 500;
    const unsigned Interval     = 10;
    const unsigned NSigma       = 1;
```

Program 1.4

```
const double   MinInit      = -3;
const double   MaxInit      = +15;
const double   SigmaInit    = 3;

const bool     PlusStrategy = false;

unsigned       i, t;

//
// initialize random number generator
//
Rng::seed( argc > 1 ?  atoi( argv[ 1 ] ) :  1234 );

//
// define populations
//
PopulationT<double> parents   ( Mu,     ChromosomeT< double >( Dimension ),
                                        ChromosomeT< double >( NSigma    ) );
PopulationT<double> offsprings( Lambda, ChromosomeT< double >( Dimension ),
                                        ChromosomeT< double >( NSigma    ) );

//
// minimization task
//
parents   .setMinimize( );
offsprings.setMinimize( );

//
// initialize parent population
//
for( i = 0; i < parents.size( ); ++i ) {
   parents[ i ][ 0 ].initialize( MinInit,   MaxInit   );
   parents[ i ][ 1 ].initialize( SigmaInit, SigmaInit );
}

//
// selection parameters (number of elitists)
//
unsigned numElitists = PlusStrategy ?  Mu :  0;

//
// standard deviations for mutation of sigma
//
double     tau0 = 1.  / sqrt( 2.  * Dimension );
double     tau1 = 1.  / sqrt( 2.  * sqrt( ( double )Dimension ) );
```

Program 1.4

```
//
// evaluate parents (only needed for elitist strategy)
//
if( PlusStrategy )
    for( i = 0; i < parents.size( ); ++i )
        parents[ i ].setFitness( ackley(parents[ i ][ 0 ] ) );

//
// iterate
//
for( t = 0; t < Iterations; ++t ) {
    //
    // generate new offsprings
    //
    for( i = 0; i < offsprings.size( ); ++i ) {
        //
        // select two random parents
        //
        Individual& mom = parents.random( );
        Individual& dad = parents.random( );

        //
        // recombine object variables discrete,
        // step sizes intermediate
        //
        offsprings[ i ][ 0 ].recombineDiscrete       ( mom[ 0 ], dad[ 0 ] );
        offsprings[ i ][ 1 ].recombineGenIntermediate( mom[ 1 ], dad[ 1 ] );

        //
        // mutate object variables normal distributed,
        // step sizes log normal distributed
        //
        offsprings[ i ][ 1 ].mutateLogNormal( tau0,  tau1 );
        offsprings[ i ][ 0 ].mutateNormal   ( offsprings[ i ][ 1 ], true );
    }

    //
    // evaluate objective function (parameters in chromosome #0)
    //
    for( i = 0; i < offsprings.size( ); ++i )
        offsprings[ i ].setFitness( ackley(offsprings[ i ][ 0 ] ) );

    //
    // select (mu,lambda) or (mu+lambda)
```

Program 1.4

```
        //
        parents.selectMuLambda( offsprings, numElitists );

        //
        // print out best value found so far
        //
        if( t % Interval == 0 )
            cout << t << "\tbest value = "
                 << parents.best( ).fitnessValue( ) << endl;
    }
}
```

Program 1.4: Canonical Evolution Strategy which minimizes Ackley's function. The flag "PlusStrategy" is used to switch between (μ, λ) and $(\mu + \lambda)$ selection.

1.5.4 Covariance Matrix Adaptation Evolution Strategy

```
#include <EALib/CMA.h>
    ⋮
// EA parameters
CMA cma;

const unsigned Iterations     = 1000;
const double   MinInit        = 1.;
const double   MaxInit        = 1.;
const double   GlobalStepInit = 1.;

unsigned Lambda             = cma.suggestLambda(Dimension);
unsigned Mu                 = cma.suggestMu(Lambda);

// define populations for minimization task
Population parents (Mu,
                    ChromosomeT< double >( Dimension ),
                    ChromosomeT< double >( Dimension ));
Population offsprings (Lambda,
                       ChromosomeT< double >( Dimension ),
                       ChromosomeT< double >( Dimension ));
offsprings.setMinimize( );
parents   .setMinimize( );

// initialize parent populations center of gravity
for(i = 0; i < parents.size( ); ++i )
  static_cast< ChromosomeT< double >& >( parents[0][0] ).
    initialize(MinInit, MaxInit);
for(j = 1; j < parents.size( ); ++j )
  static_cast< ChromosomeT< double >& >(parents[j][0]) =
    static_cast< ChromosomeT< double >& >(parents[0][0]);

// strategy parameters
vector< double > variance( Dimension );
for(i = 0; i < Dimension; i++) variance[i] = 1.;

cma.init(Dimension, variance, GlobalStepInit, parents,
         CMA::superlinear, CMA::rankmu);
// the previous three lines are equal to
//
// cma.init(Dimension, 1., parents);
//
// but demonstrates initialization with different variances
```

Program 1.5

```
    ⋮
//
// iterate
//
for( unsigned t = 0; t < Iterations; ++t ) {
  for(unsigned k = 0; k < offsprings.size( ); k++ ) {
    cma.create(offsprings[k]);
    offsprings[k].setFitness(myfitness(offsprings[k]));
  }

  // select (mu,lambda) or (mu+lambda)

  parents.selectMuLambda(offsprings, 0u);

  // update strategy parameters
  cma.updateStrategyParameters(parents);
}
    ⋮
```

Program 1.5: Covariance Matrix Adaptation Evolution Strategy (CMA-ES).

BIBLIOGRAPHY

[1] Asoh and Mühlenbein. Parallel problem solving from nature – ppsn iii. In Davidor et al. [9]. International Conference on Evolutionary Computation, The Third Conference on Parallel Problem Solving from Nature, Jerusalem, Israel, October 9-14, 1994, Proceedings.

[2] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Doctoral dissertation, University of Dortmund, Department of Computer Science, Germany, Feb. 1994.

[3] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[4] T. Bäck and F. Hoffmeister. Extended selection mechanisms in genetic algorithms. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms (ICGA'91)*, pages 92–99, University of California, San Diego, CA, USA, July 13–16 1991. Morgan Kaufmann Publishers (San Mateo, CA, 1991).

[5] J. E. Baker. Adaptive selection methods for genetic algorithms. In J. J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications (ICGA'85)*, pages 101–111, Pittsburgh, PA, July 24–26 1985. Carnegie–Mellon University, Lawrence Erlbaum Associates (Hillsdale, NJ, 1988).

[6] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In J. J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications (ICGA'87)*, pages 14–21, Cambridge, MA, July 28–31 1987. Massachusetts Institute of Technology, Lawrence Erlbaum Associates (Hillsdale, NJ, 1987).

[7] H.-G. Beyer and H.-P. Schwefel. Evolution strategies: A comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.

[8] L. B. Booker. *Intelligent Behavior as an Adaptation to the Task Environment*. Doctoral dissertation and technical report no. 243, University of Michigan, Logic of Computer Groups, Ann Arbor, 1982. Dissertation Abstracts International, 43(2), 469B. University Microfilms No. 8214966.

[9] Y. Davidor, H.-P. Schwefel, and R. Männer, editors. *Parallel Problem Solving from Nature – PPSN III*, Berlin, 1994. Springer-Verlag. International Conference on Evolutionary Computation, The Third Conference on Parallel Problem Solving from Nature, Jerusalem, Israel, October 9-14, 1994, Proceedings.

[10] L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[11] A. E. Eiben. Parallel problem solving from nature – ppsn iii. In Davidor et al. [9]. International Conference on Evolutionary Computation, The Third Conference on Parallel Problem Solving from Nature, Jerusalem, Israel, October 9-14, 1994, Proceedings.

[12] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine I ntelligence*. IEEE Press, 1995.

[13] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison–Wesley Publishing Company, Inc., Reading, MA, 1989.

[14] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In Rawlins [28], pages 69–93. The first workshop on the foundations of genetic algorithms and classifier systems (FOGA/CS-90) was held July 15-18, 1990 on the Bloomington campus of Indiana University.

[15] J. J. Grefenstette. A user's guide to *genesis*. Technical report, Navy Center for Applied Research in Artificial Intelligence, Washington, D. C., 1987.

[16] J. J. Grefenstette and J. E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In Schaffer [32], pages 20–27.

[17] N. Hansen. The CMA evolution strategy: A comparing review. In *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*. Springer-Verlag, 2006.

[18] N. Hansen, S. Müller, and P. Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). *Evolutionary Computation*, 11(1):1–18, 2003.

[19] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaption. In *Proceedings of the 1996 IEEE Intern. Conf. on Evolutionary Computation (ICEC '96)*, pages 312–317. IEEE Press, 1996.

[20] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[21] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[22] K. A. D. Jong. *An Analysis of the Behaviour of a Class of Genetic Adaptive Systems*. Ph.d. dissertation, Dept. Computer and Communication Sciences, University of Michigan, 1975. Diss. Abstr. Int. 36(10), 5140B, University Microfilms No. 76–9381.

[23] S. Kern, S. Müller, N. Hansen, D. Büche, J. Ocenasek, and P. Koumoutsakos. Learning probability distributions in continuous evolutionary algorithms – A comparative review. *Natural Computing*, 3:77–112, 2004.

[24] R. Männer and B. Manderick, editors. *Parallel Problem Solving from Nature, 2*, Amsterdam, 1992. North-Holland, (c) Elsevier Science Publishers. Proceedings of the Second Conference on Parallel Problem Solving from Nature (PPSN-92), Brussels, Belgium, 28-30 September 1992.

[25] D. Musser and A. Saini. *STL tutorial and reference guide*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, USA, 1996.

[26] A. Ostermeier. An evolution strategy with momentum adaptation of the random number distribution. In Männer and Manderick [24], pages 197–206. Proceedings of the Second Conference on Parallel Problem Solving from Nature (PPSN-92), Brussels, Belgium, 28-30 September 1992.

[27] A. Ostermeier. A derandomized approach to self adaptation of evolution strategies. *Evolutionary Computation*, 2(4):369–380, 1994.

[28] G. J. E. Rawlins, editor. *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1991. The first workshop on the foundations of genetic algorithms and classifier systems (FOGA/CS-90) was held July 15-18, 1990 on the Bloomington campus of Indiana University.

[29] I. Rechenberg. *Evolutionsstrategie '94*. Friedrich Frommann Holzboog Verlag, 1994.

[30] G. Rudolph. On correlated mutations in evolution strategies. In Männer and Manderick [24], pages 105–114. Proceedings of the Second Conference on Parallel Problem Solving from Nature (PPSN-92), Brussels, Belgium, 28-30 September 1992.

[31] G. Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Trans. Neural Networks*, 5(1):96–101, 1994.

[32] J. D. Schaffer, editor. *Proceedings of the Third International Conference on Genetic Algorithms (ICGA'89), Jul 4–7*, San Mateo, CA, 1989. George Mason University, Morgan Kaufmann Publishers, Inc.

[33] H. Schwefel. *Evolution and Optimum Seeking*. John Wiley & sons, New York, 1995.

[34] H.-P. Schwefel. *Numerische Optimierung von Computer–Modellen mittels der Evolutionsstrategie*, volume 26 of *Interdisciplinary Systems Research*. Birkhäuser, Basel, Germany, 1977.

[35] W. M. Spears and K. A. D. Jong. An analysis of multi–point crossover. In Rawlins [28], pages 301–315. The first workshop on the foundations of genetic algorithms and classifier systems (FOGA/CS-90) was held July 15-18, 1990 on the Bloomington campus of Indiana University.

[36] G. Syswerda. Uniform crossover in genetic algorithms. In Schaffer [32], pages 2–9.

[37] G. Syswerda. A study of reproduction in generational and steady state genetic algorithms. In Rawlins [28], pages 94–101. The first workshop on the foundations of genetic algorithms and classifier systems (FOGA/CS-90) was held July 15-18, 1990 on the Bloomington campus of Indiana University.

[38] L. D. Whitley. The genitor algorithm and selection pressure: Why rank–based allocation of reproductive trials is best. In Schaffer [32], pages 116–121.

[39] W. Wienholt. A refined genetic algorithm for paramter optimization problems. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA'93)*, pages 589–596, Illinois, July 17–21 1993. University of Illinois at Urbana Champaign, Morgan Kaufmann Publishers (San Mateo, CA, 1993).