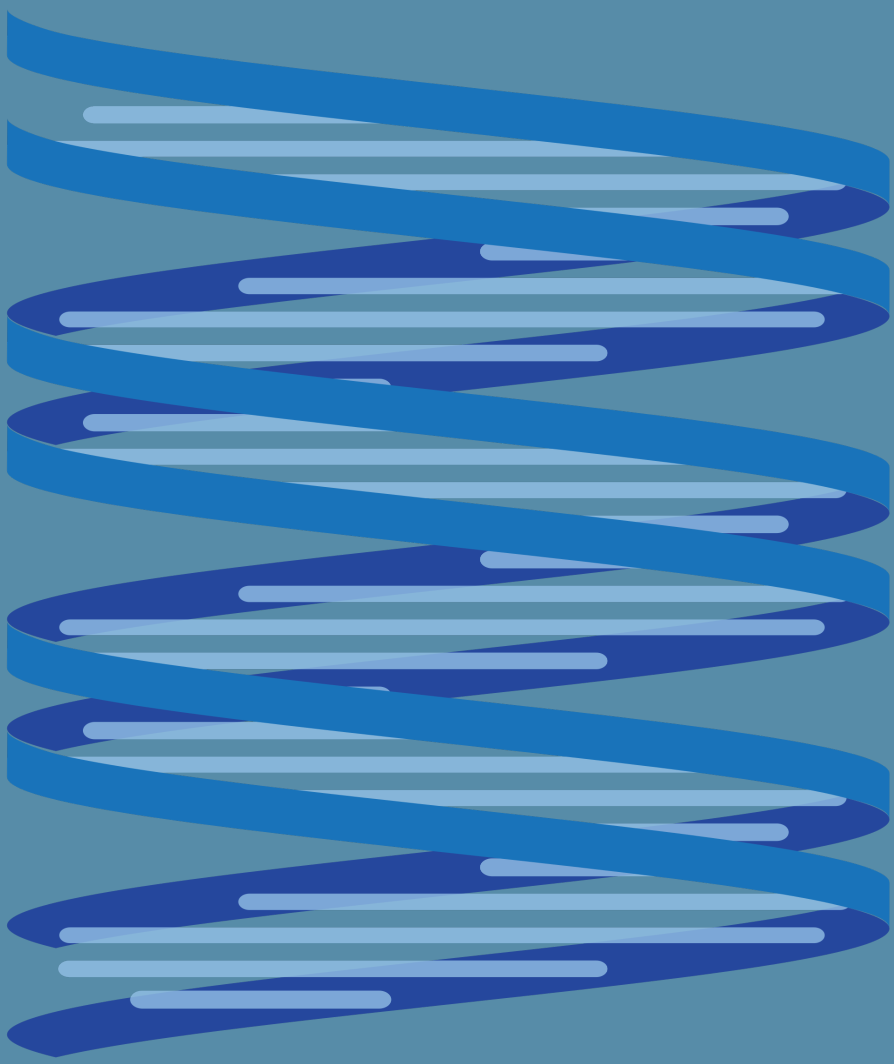
JENETICS

LIBRARY USER'S MANUAL



FRANZ WILHELMSTÖTTER

VERSION 3.8

Franz Wilhelmstötter
franz.wilhelmstoetter@gmx.at

http://jenetics.io

3.8.0—2017/04/27

Abstract

Jenetics is an **Genetic Algorithm** and **Evolutionary Algorithm** library, respectively, written in modern day Java. It is designed with a clear separation of the several algorithm concepts, e. g. `Gene`, `Chromosome`, `Genotype`, `Phenotype`, `Population` and fitness `Function`. **Jenetics** allows you to minimize or maximize the given fitness function without tweaking it. In contrast to other GA implementations, the library uses the concept of an evolution *stream* (`EvolutionStream`) for executing the evolution steps. Since the `EvolutionStream` implements the Java `Stream` interface, it works smoothly with the rest of the Java Stream API. This manual describes the concepts implemented in the **Jenetics** project and gives examples and best practice tips.

Contents

List of Figures

1 Introduction

Jenetics is a library, written in Java[1], which provides an genetic algorithm (GA) implementation. It has no runtime dependencies to other libraries, except the Java 8 runtime. Since the library is available on maven central repository[2], it can be easily integrated into existing projects. The very clear structuring of the different parts of the GA allows an easy adaption for different problem domains.

This manual is not an introduction or a tutorial for genetic and/or evolutionary algorithms in general. It is assumed that the reader has a knowledge about the structure and the functionality of genetic algorithms. Good introductions to GAs can be found in [17], [9], [16], [8], [10] or [21].

To give a first impression of the library usage, lets start with a simple »Hello World« program. This first example implements the well known bit-counting problem.

```java
import org.jenetics.BitChromosome;
import org.jenetics.BitGene;
import org.jenetics.Genotype;
import org.jenetics.engine.Engine;
import org.jenetics.engine.EvolutionResult;
import org.jenetics.util.Factory;

public final class HelloWorld {
    // 2.) Definition of the fitness function.
    private static Integer eval(final Genotype<BitGene> gt) {
        return ((BitChromosome)gt.getChromosome()).bitCount();
    }

    public static void main(final String[] args) {
        // 1.) Define the genotype (factory) suitable
        //     for the problem.
        final Factory<Genotype<BitGene>> gtf =
            Genotype.of(BitChromosome.of(10, 0.5));

        // 3.) Create the execution environment.
        final Engine<BitGene, Integer> engine = Engine
            .builder(HelloWorld::eval, gtf)
            .build();

        // 4.) Start the execution (evolution) and
        //     collect the result.
        final Genotype<BitGene> result = engine.stream()
            .limit(100)
            .collect(EvolutionResult.toBestGenotype());

        System.out.println("Hello World:\n\t" + result);
    }
}
```

Listing 1: »Hello World« GA

[1] The library is build with and depends on Java SE 8: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

[2] If you are using Gradle, you can use the following dependency string: »`io.jenetics:jenetics:3.8.0`«.

In contrast to other GA implementations, **Jenetics** uses the concept of an evolution *stream* (`EvolutionStream`) for executing the evolution steps. Since the `EvolutionStream` implements the Java `Stream` interface, it works smoothly with the rest of the Java Stream API. Now let's have a closer look at listing 1 on the preceding page and discuss this simple program step by step:

1. The probably most challenging part, when setting up a new evolution `Engine`, is to transform the problem domain into an appropriate `Genotype` (factory) representation.[3] In our example we want to count the number of *ones* of a `BitChromosome`. Since we are counting only the ones of one chromosome, we are adding only one `BitChromosome` to our `Genotype`. In general, the `Genotype` can be created with 1 to n chromosomes. For a detailed description of the genotype's structure have a look at section 3.1.3 on page 7.

2. Once this is done, the fitness function, which should be maximized, can be defined. Utilizing the new language features introduced in Java 8, we simply write a private static method, which takes the genotype we defined and calculate it's fitness value. If we want to use the optimized bit-counting method, `bitCount()`, we have to cast the `Chromosome<BitGene>` class to the actual used `BitChromosome` class. Since we know for sure that we created the `Genotype` with a `BitChromosome`, this can be done safely. A reference to the `eval` method is then used as fitness function and passed to the `Engine.build` method.

3. In the third step we are creating the *evolution* `Engine`, which is responsible for changing, respectively evolving, a given population. The Engine is highly configurable and takes parameters for controlling the evolutionary and the computational environment. For changing the evolutionary behavior, you can set different alterers and selectors (see section 3.2 on page 11). By changing the used `Executor` service, you control the number of threads, the `Engine` is allowed to use. An new `Engine` instance can only be created via its builder, which is created by calling the `Engine.builder` method.

4. In the last step, we can create a new `EvolutionStream` from our `Engine`. The `EvolutionStream` is the model (or view) of the *evolutionary* process. It serves as a »process handle« and also allows you, among other things, to control the termination of the evolution. In our example, we simply truncate the stream after 100 generations. If you don't limit the stream, the `EvolutionStream` will not terminate and run forever. The final result, the best `Genotype` in our example, is then collected with one of the predefined collectors of the `EvolutionResult` class.

As the example shows, **Jenetics** makes heavy use of the `Stream` and `Collector` classes in Java 8. Also the newly introduced lambda expressions and the functional interfaces (SAM types) play an important roll in the library design.

There are many other GA implementations out there and they may slightly differ in the order of the single execution steps. **Jenetics** uses an classical approach. Listing 2 on the following page shows the (imperative) pseudo-code of the **Jenetics** genetic algorithm steps.

[3] Section 6.1 on page 43 describes some common problem encodings.

1 $P_0 \leftarrow P_{initial}$
2 $F(P_0)$
3 **while** $!finished$ **do**
4     $g \leftarrow g + 1$
5     $S_g \leftarrow select_S(P_{g-1})$
6     $O_g \leftarrow select_O(P_{g-1})$
7     $O_g \leftarrow alter(O_g)$
8     $P_g \leftarrow filter[g_i \geq g_{max}](S_g) + filter[g_i \geq g_{max}](O_g)$
9     $F(P_g)$

Listing 2: Genetic algorithm

Line (1) creates the initial population and line (2) calculates the fitness value of the individuals. The initial population is created implicitly before the first evolution step is performed. Line (4) increases the generation number and line (5) and (6) selects the survivor and the offspring population. The offspring/survivor fraction is determined by the `offspringFraction` property of the `Engine.Builder`. The selected offspring are altered in line (7). The next line combines the survivor population and the altered offspring population—after removing the *died* individuals—to the new population. The steps from line (4) to (9) are repeated until a given termination criterion is fulfilled.

2 Architecture

The basic metaphor of the **Jenetics** library is the *Evolution Stream*, implemented via the Java 8 Stream API. Therefore it is no longer necessary (and advised) to perform the evolution steps in an *imperative* way. An evolution stream is powered by—and bound to—an *Evolution Engine*, which performs the needed *evolution* steps for each generation; the steps are described in the body of the while-loop of listing 2.

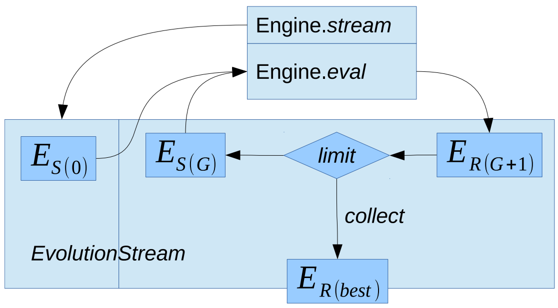


Figure 2.1: Evolution workflow

The described evolution workflow is also illustrated in figure 2.1, where $E_{S(i)}$ denotes the `EvolutionStart` object at generation i and $E_{R(i)}$ the `EvolutionResult` at the i^{th} generation. Once the evolution `Engine` is created, it can be used by multiple `EvolutionStreams`, which can be safely used in different execution threads. This is possible, because the evolution `Engine` doesn't have any mutable global state. It is practically a stateless function, $f_E : \mathrm{P} \rightarrow \mathrm{P}$, which maps a start population, P, to an evolved result population. The `Engine` function, f_E, is, of course, *non*-deterministic. Calling it twice with the same start population will lead to different result populations.

The evolution process terminates, if the `EvolutionStream` is truncated and the `EvolutionStream` truncation is controlled by the `limit` predicate. As long as the predicate returns true, the evolution is continued.[4] At last, the `EvolutionResult` is collected from the `EvolutionStream` by one of the available `EvolutionResult` collectors.

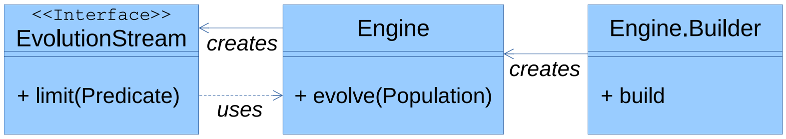


Figure 2.2: Evolution engine model

Figure 2.2 shows the *static* view of the main *evolution* classes, together with its dependencies. Since the `Engine` class itself is immutable, and can't be changed after creation, it is instantiated (configured) via a builder. The `Engine` can be used to create an arbitrary number of `EvolutionStreams`. The `EvolutionStream` is used to control the evolutionary process and collect the final result. This is done in the same way as for the normal `java.util.stream.Stream` classes. With the additional `limit(Predicate)` method, it is possible to truncate the `EvolutionStream` if some termination criteria is fulfilled. The separation of `Engine` and `EvolutionStream` is the separation of the evolution definition and evolution execution.

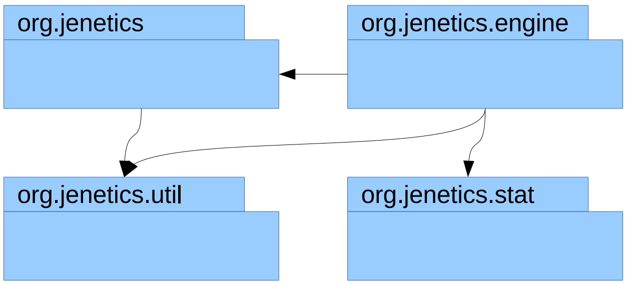


Figure 2.3: Package structure

In figure 2.3 the package structure of the library is shown and it consists of the following packages:

`org.jenetics` This is the base package of the **Jenetics** library and contains all domain classes, like `Gene`, `Chromosome` or `Genotype`. Most of this types are immutable data classes and doesn't implement any behavior. It also contains the `Selector` and `Alterer` interfaces and its implementations. The classes in this package are (almost) sufficient to implement an own GA.

`org.jenetics.engine` This package contains the actual GA implementation classes, e. g. `Engine`, `EvolutionStream` or `EvolutionResult`. They mainly operate on the domain classes of the `org.jenetics` package.

[4]See section 6.5 on page 57 for a detailed description of the available termination strategies.

org.jenetics.stat This package contains additional statistics classes which are not available in the Java core library. Java only includes classes for calculating the sum and the average of a given *numeric* stream (e. g. DoubleSummaryStatistics). With the additions in this package it is also possible to calculate the variance, skewness and kurtosis—using the DoubleMomentStatistics class. The EvolutionStatistics object, which can be calculated for every generation, relies on the classes of this package.

org.jenetics.util This package contains the collection classes (Seq, ISeq and MSeq) which are used in the public interfaces of the Chromosome and Genotype. It also contains the RandomRegistry class, which implements the global PRNG lookup, as well as helper IO classes for serializing Genotypes and whole Populations.

3 Base classes

This chapter describes the main classes which are needed to setup and run an genetic algorithm with the **Jenetics**[5] library. They can roughly divided into three types:

Domain classes This classes form the domain model of the evolutionary algorithm and contain the structural classes like Gene and Chromosome. They are located in the org.jenetics package.

Operation classes This classes operates on the domain classes and includes the Alterer and Selector classes. They are also located in the org.jenetics package.

Engine classes This classes implements the actual evolutionary algorithm and reside solely in the org.jenetics.engine package.

3.1 Domain classes

Most of the domain classes are pure data classes and can be treated as *value* objects[6]. All Gene and Chromosome implementations are immutable as well as the Genotype and Phenotype class. The only exception is the Population class, where it is possible to add and/or remove elements after it's creation.

Figure 3.1 on the following page shows the class diagram of the domain classes. All domain classes are located in the org.jenetics package. The Gene is the base of the class structure. Genes are aggregated in Chromosomes. One to n Chromosomes are aggregated in Genotypes. A Genotype and a fitness Function form the Phenotype, which are collected into a Population.

3.1.1 Gene

Genes are the basic building blocks of the **Jenetics** library. They contain the actual information of the encoded solution, the allele. Some of the implementations also contains domain information of the *wrapped* allele. This is the case

[5] The documentation of the whole API is part of the download package or can be viewed online: http://jenetics.io/javadoc/org.jenetics/3.8/index.html.
[6] https://en.wikipedia.org/wiki/Value_object

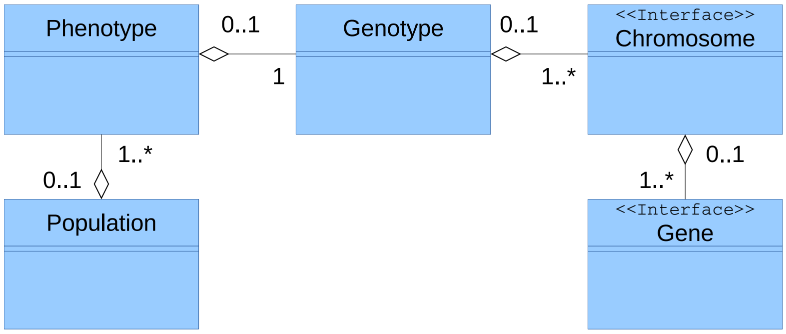


Figure 3.1: Domain model

for all `BoundedGene`, which contain the allowed minimum and maximum values. All `Gene` implementations are final and immutable. In fact, they are all value-based classes and fulfill the properties which are described in the Java 8 API documentation[13].[7]

Beside the container functionality for the allele, every `Gene` is its own factory and is able to create new, random instances of the same type and with the same constraints. The factory methods are used by the `Alterer`s for creating new `Gene`s from the existing one and play a crucial role by the exploration of the problem space.

```java
public interface Gene<A, G extends Gene<A, G>>
    extends Factory<G>, Verifiable
{
    public A getAllele();
    public G newInstance();
    public G newInstance(A allele);
    public boolean isValid();
}
```

Listing 3: `Gene` interface

Listing 3 shows the most important methods of the `Gene` interface. The `isValid` method, introduced by the `Verifiable` interface, allows the gene to mark itself as invalid. All invalid genes are replaced with new ones during the evolution phase.

The available `Gene` implementations in the **Jenetics** library should cover a wide range of problem encodings. Refer to chapter 5.1 on page 37 for how to implement your own `Gene` types.

3.1.2 Chromosome

A `Chromosome` is a collection of `Gene`s which contains at least one `Gene`. This allows to encode problems which requires more than one `Gene`. Like the `Gene` interface, the `Chromosome` is also it's own factory and allows to create a new `Chromosome` from a given `Gene` sequence.

```java
public interface Chromosome<G extends Gene<?, G>>
    extends Factory<Chromosome<G>>, Iterable<G>, Verifiable
```

[7]It is also worth reading the blog entry from Stephen Colebourne: `http://blog.joda.org/2014/03/valjos-value-java-objects.html`

```java
{
    public Chromosome<G> newInstance(ISeq<G> genes);
    public G getGene(int index);
    public ISeq<G> toSeq();
    public Stream<G> stream();
    public int length();
}
```

Listing 4: `Chromosome` interface

Listing 4 on the previous page shows the main methods of the `Chromosome` interface. This are the methods for accessing single `Genes` by index and as an `ISeq` respectively, and the factory method for creating a new `Chromosome` from a given sequence of `Genes`. The factory method is used by the `Alterer` classes which were able to create altered `Chromosome` from a (changed) `Gene` sequence.

3.1.3 Genotype

The central class, the evolution `Engine` is working with, is the `Genotype`. It is the *structural* and immutable representative of an individual and consists of one to n `Chromosomes`. All `Chromosomes` must be parameterized with the same `Gene` type, but it is allowed to have different lengths and constraints. The allowed minimal- and maximal values of a `NumericChromosome` is an example of such a constraint. Within the same chromosome, all numeric gene alleles must lay within the defined minimal- and maximal values.

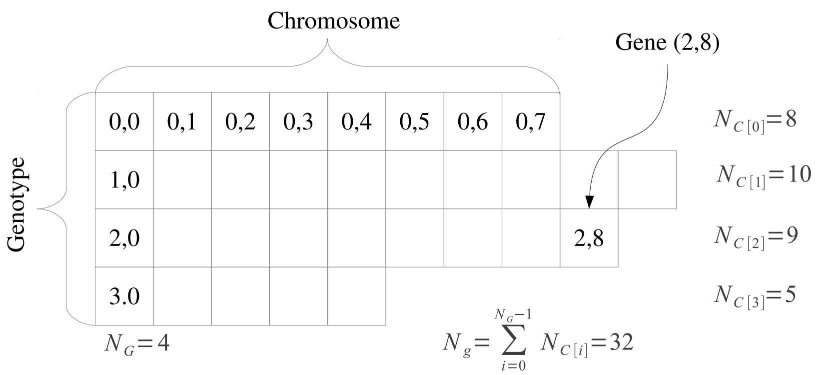


Figure 3.2: `Genotype` structure

Figure 3.2 shows the `Genotype` structure. A `Genotype` consists of N_G `Chromosomes` and a `Chromosome` consists of $N_{C[i]}$ `Genes` (depending on the `Chromosome`). The overall number of `Genes` of a `Genotype` is given by the sum of the `Chromosome`'s `Genes`, which can be accessed via the `Genotype.getNumberOfGenes()` method:

$$N_g = \sum_{i=0}^{N_G-1} N_{C[i]} \tag{3.1}$$

As already mentioned, the `Chromosomes` of a `Genotype` doesn't have to have necessarily the same size. It is only required that all genes are from the same type and the Genes within a Chromosome have the same constraints; e. g. the same min- and max values for numerical Genes.

```
Genotype<DoubleGene> genotype = Genotype.of(
    DoubleChromosome.of(0.0,  1.0,  8),
    DoubleChromosome.of(1.0,  2.0, 10),
    DoubleChromosome.of(0.0, 10.0,  9),
    DoubleChromosome.of(0.1,  0.9,  5)
);
```

The code snippet in the listing above creates a Genotype with the same structure as shown in figure 3.2 on the previous page. In this example the DoubleGene has been chosen as Gene type.

Genotype vector The Genotype is essentially a two-dimensional composition of Genes. This makes it trivial to create Genotypes which can be treated as a Gene matrices. If its needed to create a vector of Genes, there are two possibilities to do so:

1. creating a *row-major* or
2. creating a *column-major*

Genotype vector. Each of the two possibilities have specific advantages and disadvantages.

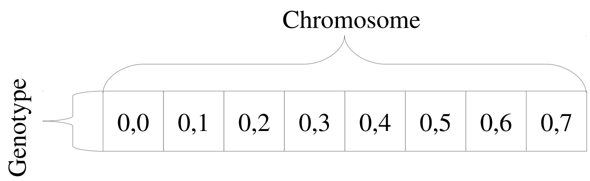


Figure 3.3: Row-major Genotype vector

Figure 3.3 shows a Genotype vector in row-major layout. A Genotype vector of length n needs one Chromosome of length n. Each Gene of such a vector obeys the same constraints. E. g., for Genotype vectors containing NumericGenes, all Genes must have the same minimum and maximum values. If the problem space doesn't need to have different minimum and maximum values, the row-major Genotype vector is the preferred choice. Beside the easier Genotype creation, the available Recombinator alterers are more efficient in exploring the search domain.

If the problem space allows equal Gene constraint, the row-major Genotype vector encoding should be chosen. It is easier to create and the available Recombinator classes are more efficient in exploring the search domain.

The following code snippet shows the creation of a row-major Genotype vector. All Alterers derived from the Recombinator do a fairly good job in exploring the problem space for row-major Genotype vector.

```
Genotype<DoubleGene> genotype = Genotype.of(
    DoubleChromosome.of(0.0, 1.0, 8)
);
```

The column-major Genotype vector layout must be chosen when the problem space requires components (Genes) with different constraints. This is almost the *only* reason for choosing the column-major layout. The layout of this Genotype vector is shown in 3.4. For a vector of length n, n Chromosomes of length *one* are needed.

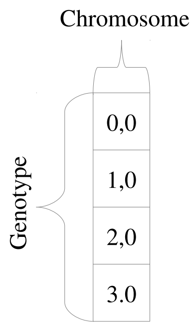


Figure 3.4: Column-major Genotype vector

The code snippet below shows how to create a Genotype vector in column-major layout. It's a little more effort to create such a vector, since every Gene has to be wrapped into a separate Chromosome. The DoubleChromosome in the given example has length of one, when the length parameter is omitted.

```
Genotype<DoubleGene> genotype = Genotype.of(
    DoubleChromosome.of(0.0, 1.0),
    DoubleChromosome.of(1.0, 2.0),
    DoubleChromosome.of(0.0, 10.0),
    DoubleChromosome.of(0.1, 0.9)
);
```

The greater flexibility of a column-major Genotype vector has to be payed with a lower exploration capability of the Recombinator alterers. Using Crossover alterers will have the same effect as the SwapMutator, when used with row-major Genotype vectors. Recommended alterers for vectors of NumericGenes are:

- MeanAlterer[8],
- LineCrossover[9] and
- IntermediateCrossover[10]

See also 6.2.2 on page 51 for an advanced description on how to use the predefined vector codecs.

[8] See 3.2.2 on page 19.
[9] See 3.2.2 on page 19.
[10] See 3.2.2 on page 20.

`Genotype` scalar A very special case of a `Genotype` contains only one `Chromosome` with length one. The layout of such a `Genotype` scalar is shown in 3.5. Such `Genotype`s are mostly used for encoding real function problems.

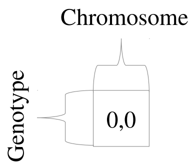


Figure 3.5: `Genotype` scalar

How to create a `Genotype` for a real function optimization problem, is shown in the code snippet below. The recommended `Alterer`s are the same as for column-major `Genotype` vectors: `MeanAlterer`, `LineCrossover` and `IntermediateCrossover`.

```
Genotype<DoubleGene> genotype = Genotype.of(
    DoubleChromosome.of(0.0, 1.0)
);
```

See also 6.2.1 on page 50 for an advanced description on how to use the predefined scalar codecs.

3.1.4 Phenotype

The `Phenotype` is the *actual* representative of an individual and consists of the `Genotype` and the fitness `Function`, which is used to (lazily) calculate the `Genotype`'s fitness value.[11] It is *only* a container which forms the *environment* of the `Genotype` and doesn't change the structure. Like the `Genotype`, the `Phenotype` is immutable and can't be changed after creation.

```
public final class Phenotype<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
    implements Comparable<Phenotype<G, C>>
{
    public C getFitness();
    public Genotype<G> getGenotype();
    public long getAge(long currentGeneration);
    public void evaluate();
}
```

Listing 5: `Phenotype` class

Listing 5 shows the main methods of the `Phenotype`. The `fitness` property will return the actual fitness value of the `Genotype`, which can be fetched with the `getGenotype` method. To make the runtime behavior predictable, the fitness value is evaluated lazily. Either by querying the `fitness` property or through the call of the `evaluate` method. The evolution `Engine` is calling the evaluate method in a separate step and makes the fitness evaluation time available through the `EvolutionDurations` class. Additionally to the fitness value, the

[11] Since the fitness `Function` is shared by all `Phenotype`s, calls to the fitness `Function` must be idempotent. See section 3.3.1 on page 20.

`Phenotype` contains the generation when it was created. This allows to calculate the current age and the removal of overaged individuals from the `Population`.

3.1.5 Population

The end of the class hierarchy of the domain model is the `Population`. It is a collection of individuals and forms the start and the end of an evolution step.

```java
public final class Population<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
    implements List<Phenotype<G, C>>
{
    public Phenotype<G, C> get(int index);
    public void add(Phenotype<G, C> phenotype);
    public void sortWith(Comparator<? super C> comparator);
}
```

Listing 6: `Population` class

Listing 6 gives on overview of the most important methods of the `Population` class. In addition to the `List` methods, it provides a method for sorting the `Phenotype`s. Some `Selector` implementations require a sorted list of individuals according its fitness value. Calling `population.sortWith(optimize-.descending())` will sort the `Population`, so that the first element will be the individual with the best fitness.

3.2 Operation classes

Genetic operators are used for creating *genetic* diversity (`Alterer`) and selecting potentially useful solutions for recombination (`Selector`). This section gives an overview about the genetic operators available in the **Jenetics** library. It also contains some *theoretical* information, which should help you to choose the right combination of operators and parameters, for the problem to be solved.

3.2.1 Selector

Selectors are responsible for selecting a given number of individuals from the population. The selectors are used to divide the population into *survivors* and *offspring*. The selectors for offspring and for the survivors can be chosen independently.

The selection process of the **Jenetics** library acts on `Phenotype`s and indirectly, via the fitness function, on `Genotype`s. Direct `Gene`- or `Population` selection is not supported by the library.

```java
Engine<DoubleGene, Double> engine = Engine.builder(...)
    .offspringFraction(0.7)
    .survivorsSelector(new RouletteWheelSelector<>())
    .offspringSelector(new TournamentSelector<>())
    .build();
```

The `offspringFraction`, $f_O \in [0, 1]$, determines the number of selected offspring

$$N_{O_g} = \|O_g\| = \mathrm{rint}\left(\|P_g\| \cdot f_O\right) \tag{3.2}$$

and the number of selected survivors

$$N_{S_g} = \|S_g\| = \|P_g\| - \|O_g\| \,. \tag{3.3}$$

The **Jenetics** library contains the following selector implementations:

- `TournamentSelector`
- `TruncationSelector`
- `MonteCarloSelector`
- `ProbabilitySelector`
- `RouletteWheelSelector`
- `LinearRankSelector`
- `ExponentialRankSelector`
- `BoltzmannSelector`
- `StochasticUniversalSelector`

Beside the well known standard selector implementation the `ProbabilitySelector` is the base of a set of fitness proportional selectors.

Tournament selector In tournament selection the best individual from a random sample of s individuals is chosen from the population Pg. The samples are drawn with replacement. An individual will win a tournament only if the fitness is greater than the fitness of the other $s - 1$ competitors. Note that the worst individual never survives, and the best individual wins in all the tournaments it participates. The selection pressure can be varied by changing the tournament size$s$. For large values of s, weak individuals have less chance of being selected.

Truncation selector In truncation selection individuals are sorted according to their fitness. (This is one of the selectors, which relies on the `sortWith` method of the `Population` class.) Only the n best individuals are selected. The truncation selection is a very basic selection algorithm. It has it's strength in fast selecting individuals in large populations, but is not very often used in practice.

Monte Carlo selector The Monte Carlo selector selects the individuals from a given population randomly. This selector can be used to measure the performance of a other selectors. In general, the performance of a selector should be better than the selection performance of the Monte Carlo selector.

Probability selectors Probability selectors are a variation of *fitness proportional* selectors and selects individuals from a given population based on it's *selection* probability $P(i)$. Fitness proportional selection works as shown in figure 3.6. An uniform distributed random number $r \in [0, F)$ specifies which individual is selected, by argument minimization:

$$i \leftarrow \underset{n \in [0,N)}{\mathrm{argmin}} \left\{ r < \sum_{i=0}^{n} f_i \right\}, \tag{3.4}$$

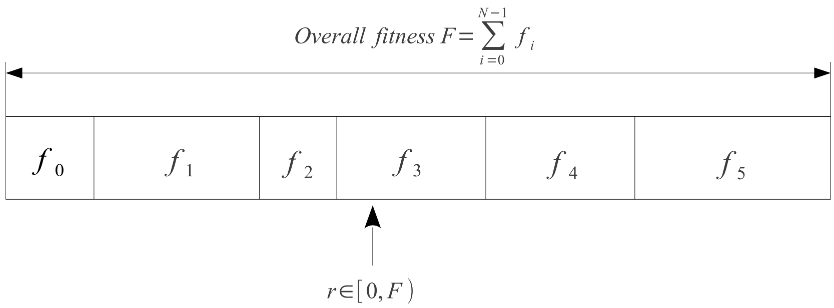


Figure 3.6: Fitness proportional selection

where N is the number of individuals and f_i the fitness value of the i^{th} individual. The probability selector works the same way, only the fitness value f_i is replaced by the individual's selection probability $P(i)$. It is not necessary to sort the population. The selection probability of an individual i follows a binomial distribution

$$P(i,k) = \begin{pmatrix} n \\ k \end{pmatrix} P(i)^k \left(1 - P(i)\right)^{n-k} \tag{3.5}$$

where n is the overall number of selected individuals and k the number of individual$i$ in the set of selected individuals. The runtime complexity of the implemented probability selectors is $O(n + \log(n))$ instead of $O(n^2)$ as for the naive approach: *A binary (index) search is performed on the summed probability array.*

Roulette-wheel selector The roulette-wheel selector is also known as fitness proportional selector. In the **Jenetics** library it is implemented as *probability* selector. The fitness value f_i is used to calculate the selection probability of individual i.

$$P(i) = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{3.6}$$

Selecting n individuals from a given population is equivalent to play n times on the roulette-wheel. The population don't have to be sorted before selecting the individuals. Roulette-wheel selection is one of the traditional selection strategies.

Linear-rank selector In linear-ranking selection the individuals are sorted according to their fitness values. The rank N is assigned to the best individual and the rank 1 to the worst individual. The selection probability $P(i)$ of individual i is linearly assigned to the individuals according to their rank.

$$P(i) = \frac{1}{N}\left(n^- + \left(n^+ - n^-\right)\frac{i-1}{N-1}\right). \tag{3.7}$$

Here $\frac{n^-}{N}$ is the probability of the worst individual to be selected and $\frac{n^+}{N}$ the probability of the best individual to be selected. As the population size is held constant, the condition $n^+ = 2 - n^-$ and $n^- \geq 0$ must be fulfilled. Note that all individuals get a different rank, respectively a different selection probability, even if they have the same fitness value.[5]

Exponential-rank selector An alternative to the *weak* linear-rank selector is to assign survival probabilities to the sorted individuals using an exponential function:

$$P(i) = (c-1)\,\frac{c^{i-1}}{c^N - 1}, \tag{3.8}$$

where c must within the range $[0\ldots1)$. A small value of c increases the probability of the best individual to be selected. If c is set to zero, the selection probability of the best individual is set to one. The selection probability of all other individuals is zero. A value near one equalizes the selection probabilities. This selector sorts the population in descending order before calculating the selection probabilities.

Boltzmann selector The selection probability of the Boltzmann selector is defined as

$$P(i) = \frac{\mathrm{e}^{b\cdot f_i}}{Z}, \tag{3.9}$$

where b is a parameter which controls the selection intensity and Z is defined as

$$Z = \sum_{i=1}^{n} \mathrm{e}^{f_i}. \tag{3.10}$$

Positive values of b increases the selection probability of individuals with high fitness values and negative values of b decreases it. If b is zero, the selection probability of all individuals is set to $\frac{1}{N}$.

Stochastic-universal selector Stochastic-universal selection [1] (SUS) is a method for selecting individuals according to some given probability in a way that minimizes the chance of fluctuations. It can be viewed as a type of roulette game where we now have p equally spaced points which we spin. SUS uses a single random value for selecting individuals by choosing them at equally spaced intervals. The selection method was introduced by James Baker. [2] Figure 3.7

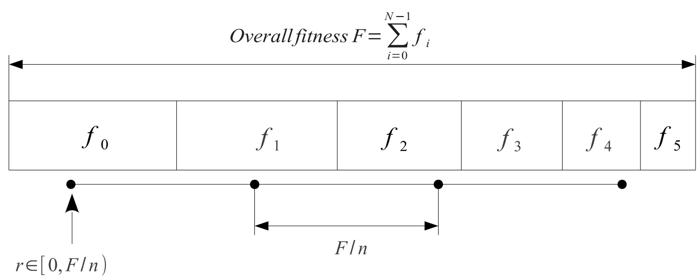


Figure 3.7: Stochastic-universal selection

shows the function of the stochastic-universal selection, where n is the number of individuals to select. Stochastic universal sampling ensures a selection of offspring, which is closer to what is deserved than roulette wheel selection.[17]

3.2.2 Alterer

The problem encoding (representation) determines the bounds of the search space, but the `Alterer`s determine how the space can be traversed: `Alterer`s are responsible for the genetic diversity of the `EvolutionStream`. The two `Alterer` types used in **Jenetics** are:

1. mutation and
2. recombination (e. g. crossover).

First we will have a look at the mutation — There are two distinct roles *mutation* plays in the evolution process:

1. **Exploring the search space**: By making small moves, mutation allows a population to explore the search space. This exploration is often slow compared to crossover, but in problems where crossover is disruptive this can be an important way to explore the landscape.
2. **Maintaining diversity**: Mutation prevents a population from correlating. Even if most of the search is being performed by crossover, mutation can be vital to provide the diversity which crossover needs.

The mutation probability, $P(m)$, is the parameter that must be optimized. The optimal value of the mutation rate depends on the role mutation plays. If mutation is the only source of exploration (if there is no crossover), the mutation rate should be set to a value that ensures that a reasonable neighborhood of solutions is explored.

The mutation probability, $P(m)$, is defined as the probability that a specific gene, over the whole population, is mutated. That means, the (average) number of genes mutated by a mutator is

$$\hat{\mu} = N_P \cdot N_g \cdot P(m) \tag{3.11}$$

where N_g is the number of available genes of a genotype and N_P the population size (revere to equation 3.1 on page 7).

Mutator The mutator has to deal with the problem, that the genes are arranged in a $3D$ structure (see chapter 3.1.3). The mutator selects the gene which will be mutated in three steps:

1. Select a genotype $G[i]$ from the population with probability $P_G(m)$,
2. select a chromosome $C[j]$ from the selected genotype $G[i]$ with probability $P_C(m)$ and
3. select a gene $g[k]$ from the selected chromosome $C[j]$ with probability $P_g(m)$.

The needed *sub*-selection probabilities are set to

$$P_G(m) = P_C(m) = P_g(m) = \sqrt[3]{P(m)}. \tag{3.12}$$

Gaussian mutator The Gaussian mutator performs the mutation of number genes. This mutator picks a new value based on a Gaussian distribution around the current value of the gene. The variance of the new value (before clipping to the allowed gene range) will be

$$\hat{\sigma}^2 = \left(\frac{g_{max} - g_{min}}{4}\right)^2 \tag{3.13}$$

where g_{min} and g_{max} are the valid minimum and maximum values of the number gene. The new value will be cropped to the gene's boundaries.

Swap mutator The swap mutator changes the order of genes in a chromosome, with the hope of bringing related genes closer together, thereby facilitating the production of building blocks. This mutation operator can also be used for combinatorial problems, where no duplicated genes within a chromosome are allowed, e. g. for the TSP.

The second alterer type is the recombination — An enhanced genetic algorithm (EGA) combine elements of existing solutions in order to create a new solution, with some of the properties of each parents. Recombination creates a new chromosome by combining parts of two (or more) parent chromosomes. This combination of chromosomes can be made by selecting one or more crossover points, splitting these chromosomes on the selected points, and merge those portions of different chromosomes to form new ones.

```java
void recombine(final Population<G, C> pop) {
    // Select the Genotypes for crossover.
    final Random random = RandomRegistry.getRandom();
    final int i1 = random.nextInt(pop.length());
    final int i2 = random.nextInt(pop.length());
    final Phenotype<G, C> pt1 = pop.get(i1);
    final Phenotype<G, C> pt2 = pop.get(2);
    final Genotype<G> gt1 = pt1.getGenotype();
    final Genotype<G> gt2 = pt2.getGenotype();

    //Choosing the Chromosome for crossover.
    final int chIndex =
        random.nextInt(min(gt1.length(), gt2.length()));
    final MSeq<Chromosome<G>> c1 = gt1.toSeq().copy();
    final MSeq<Chromosome<G>> c2 = gt2.toSeq().copy();
    final MSeq<G> genes1 = c1.get(chIndex).toSeq().copy();
    final MSeq<G> genes2 = c2.get(chIndex).toSeq().copy();

    // Perform the crossover.
    crossover(genes1, genes2);
    c1.set(chIndex, c1.get(chIndex).newInstance(genes1.toISeq()));
    c2.set(chIndex, c2.get(chIndex).newInstance(genes2.toISeq()));

    //Creating two new Phenotypes and replace the old one.
    pop.set(i1, pt1.newInstance(gt1.newInstance(c1.toISeq())));
    pop.set(i2, pt2.newInstance(gt1.newInstance(c2.toISeq())));
}
```

Listing 7: Chromosome selection for recombination

Listing 7 on the preceding page shows how two chromosomes are selected for *recombination*. It is done this way for preserving the given *constraints* and to avoid the creation of invalid individuals.

Because of the possible different `Chromosome` length and/or `Chromosome` constraints within a `Genotype`, only `Chromosomes` with the same `Genotype` position are recombined (see listing 7 on the previous page).

The recombination probability, $P(r)$, determines the probability that a given individual (genotype) of a population is selected for recombination. The (mean) number of changed individuals depend on the concrete implementation and can be vary from $P(r) \cdot N_G$ to $P(r) \cdot N_G \cdot O_R$, where O_R is the order of the recombination, which is the number of individuals involved in the `combine` method.

Single-point crossover The single-point crossover changes two children chromosomes by taking two chromosomes and cutting them at some, randomly chosen, site. If we create a child and its complement we preserve the total number of genes in the population, preventing any genetic drift. Single-point crossover is the classic form of crossover. However, it produces very slow mixing compared with multi-point crossover or uniform crossover. For problems where the site position has some intrinsic meaning to the problem single-point crossover can lead to smaller disruption than multiple-point or uniform crossover.

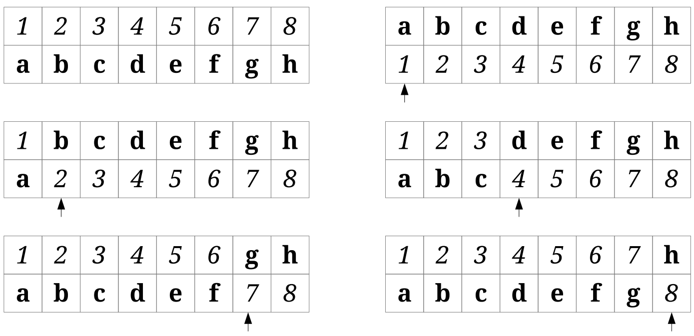


Figure 3.8: Single-point crossover

Figure 3.8 shows how the `SinglePointCrossover` class is performing the crossover for different crossover points—in the given example for the chromosome indexes 0, 1, 3, 6 and 7.

Multi-point crossover If the `MultiPointCrossover` class is created with one crossover point, it behaves exactly like the single-point crossover. The following picture shows how the multi-point crossover works with two crossover points, defined at index 1 and 4.

Figure 3.10 you can see how the crossover works for an odd number of crossover points.

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

a	**b**	**c**	**d**	*5*	*6*	*7*	*8*
1	*2*	*3*	*4*	**e**	**f**	**g**	**h**

1	*2*	*3*	**d**	**e**	**f**	*7*	*8*
a	**b**	**c**	*4*	*5*	*6*	**g**	**h**

a	**b**	**c**	**d**	**e**	**f**	**g**	*8*
1	*2*	*3*	*4*	*5*	*6*	*7*	**h**

Figure 3.9: 2-point crossover

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

a	**b**	**c**	**d**	*5*	*6*	*7*	**h**
1	*2*	*3*	*4*	**e**	**f**	**g**	*8*

1	*2*	*3*	**d**	**e**	*6*	**g**	**h**
a	**b**	**c**	*4*	*5*	**f**	*7*	*8*

a	**b**	**c**	*4*	*5*	**f**	**g**	**h**
1	*2*	*3*	**d**	**e**	*6*	*7*	*8*

Figure 3.10: 3-point crossover

Partially-matched crossover The partially-matched crossover guarantees that all genes are found exactly once in each chromosome. No gene is duplicated by this crossover strategy. The partially-matched crossover (PMX) can be applied usefully in the TSP or other permutation problem encodings. Permutation encoding is useful for all problems where the fitness only depends on the ordering of the genes within the chromosome. This is the case in many combinatorial optimization problems. Other crossover operators for combinatorial optimization are:

- order crossover
- cycle crossover
- edge recombination crossover
- edge assembly crossover

The PMX is similar to the two-point crossover. A crossing region is chosen by selecting two crossing points (see figure 3.11 *a*)). After performing the

a)

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
9	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

b)

0	*1*	*2*	**6**	**5**	**4**	*6*	*7*	*8*	*9*
9	**8**	**7**	*3*	*4*	*5*	**3**	**2**	**1**	**0**

c)

0	*1*	*2*	**6**	**5**	**4**	**3**	*7*	*8*	*9*
9	**8**	**7**	*3*	*4*	*5*	*6*	**2**	**1**	**0**

Figure 3.11: Partially-matched crossover

crossover we–normally–got two invalid chromosomes (figure 3.11 *b*)). Chromosome 1 contains the value 6 twice and misses the value 3. On the other side chromosome 2 contains the value 3 twice and misses the value 6. We can observe that this crossover is equivalent to the exchange of the values 3→6, 4→5 and 5→4. To repair the two chromosomes we have to apply this exchange outside the crossing region (figure 3.11 *b*)). At the end figure 3.11 *c*) shows the repaired chromosome.

Uniform crossover In uniform crossover, the genes at index i of two chromosomes are swapped with the swap-probability, p_S. Empirical studies shows that uniform crossover is a more exploitative approach than the traditional exploitative approach that maintains longer schemata. This leads to a better search of the design space with maintaining the exchange of good information.[6]

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

a	*2*	**c**	*4*	*5*	**f**	**g**	*8*
1	**b**	*3*	**d**	**e**	*6*	*7*	**h**

Figure 3.12: Uniform crossover

Figure 3.12 shows an example of a uniform crossover with four crossover points. A gene is swapped, if a uniformly created random number, $r \in [0, 1]$, is smaller than the swap-probability, p_S. The following code snippet shows how these swap indexes are calculated, in a functional way.

```java
final Random random = RandomRegistry.getRandom();
final int length = 8;
final double ps = 0.5;
final int[] indexes = IntRange.range(0, length)
    .filter(i -> random.nextDouble() < ps)
    .toArray();
```

Mean alterer The Mean alterer works on genes which implement the `Mean` interface. All numeric genes implement this interface by calculating the arithmetic mean of two genes.

Line crossover The line crossover[12] takes two *numeric* chromosomes and treats it as a real number vector. Each of this vectors can also be seen as a point in $\mathbb{R}^n$. If we draw a line through this two points (chromosome), we have the possible values of the new chromosomes, which all lie on this line.

Figure 3.13 on the next page shows how the two chromosomes form the two three-dimensional vectors (black circles). The dashed line, connecting the two points, form the possible solutions created by the line crossover. An additional variable, p, determines how far out along the line the created children will be. If $p = 0$ then the children will be located along the line within the hypercube. If $p > 0$, the children may be located on an arbitrary place on the line, even

[12]The line crossover, also known as line recombination, was originally described by Heinz Mühlenbein and Dirk Schlierkamp-Voosen.[11]

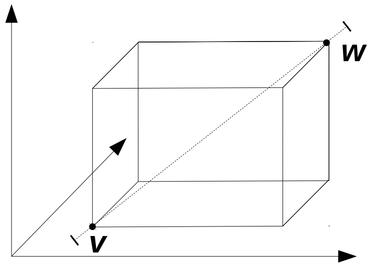


Figure 3.13: Line crossover hypercube

outside of the hypercube.This is useful if you want to explore *unknown* regions, and you need a way to generate chromosomes further out than the parents are.

The *internal* random parameters, which define the location of the new crossover point, are generated once for the whole vector (chromosome). If the `LineCrossover` generates numeric genes which lie outside the allowed minimum and maximum value, it simply uses the original gene and rejects the generated, invalid one.

Intermediate crossover The intermediate crossover is quite similar to the line crossover. It differs in the way on how the *internal* random parameters are generated and the handling of the invalid–out of range–genes. The *internal* random parameters of the `IntermediateCrossover` class are generated for *each* gene of the chromosome, instead once for all genes. If the newly generated gene is not within the allowed range, a new one is created. This is repeated, until a valid gene is built.

The crossover parameter, p, has the same properties as for the line crossover. If the chosen value for p is greater than 0, it is likely that some genes must be created more than once, because they are not in the valid range. The probability for gene re-creation rises sharply with the value of p. Setting p to a value greater than one, doesn't make sense in most of the cases. A value greater than 10 should be avoided.

3.3 Engine classes

The *executing* classes, which perform the actual evolution, are located in the `org.jenetics.engine` package. The *evolution stream* (`EvolutionStream`) is the base metaphor for performing an GA. On the `EvolutionStream` you can define the termination predicate and than collect the final `EvolutionResult`. This decouples the static data structure from the executing evolution part. The `EvolutionStream` is also very flexible, when it comes to collecting the final result. The `EvolutionResult` class has several predefined collectors, but you are free to create your own one, which can be seamlessly *plugged* into the existing stream.

3.3.1 Fitness function

The fitness `Function` is also an important part when modeling an genetic algorithm. It takes a `Genotype` as argument and returns, at least, a `Comparable` object as result—the fitness value. This allows the evolution `Engine`, respectively the selection operators, to select the offspring- and survivor `Population`. Some selectors have stronger requirements to the fitness value than a `Comparable`, but this constraints is checked by the Java type system at compile time.

Since the fitness `Function` is shared by all `Phenotypes`, calls to the fitness `Function` has to be idempotent. A fitness `Function` is idempotent if, whenever it is applied twice to any `Genotype`, it returns the same fitness value as if it were applied once. In the simplest case, this is achieved by `Functions` which doesn't contain any global *mutable* state.

The following example shows the simplest possible fitness `Function`. This `Function` simply returns the allele of a 1x1 *float* `Genotype`.

```java
public class Main {
    static Double identity(final Genotype<DoubleGene> gt) {
        return gt.getGene().getAllele();
    }

    public static void main(final String[] args) {
        // Create fitness function from method reference.
        Function<Genotype<DoubleGene>, Double>> ff1 =
            Main::identity;

        // Create fitness function from lambda expression.
        Function<Genotype<DoubleGene>, Double>> ff2 = gt ->
            gt.getGene().getAllele();
    }
}
```

The first type parameter of the `Function` defines the kind of `Genotype` from which the fitness value is calculated and the second type parameter determines the return type, which must be, at least, a `Comparable` type.

3.3.2 Fitness scaler

The fitness value, calculated by the fitness `Function`, is treated as the *raw*-fitness of an individual. The **Jenetics** library allows you to apply an additional scaling function on the raw-fitness to form the fitness value which is used by the selectors. This can be useful when using probability selectors (see chapter 3.2.1 on page 12), where the actual amount of the fitness value influences the selection probability. In such cases, the fitness scaler gives you additional flexibility when selecting offspring and survivors. In the default configuration the raw-fitness is equal to the actual fitness value, that means, the used fitness scaler is the identity function.

```java
class Main {
    public static void main(final String[] args) {
        Engine<DoubleGene, Double> engine = Engine.builder(...)
            .fitnessScaler(Math::sqrt)
            .build();
    }
}
```

The given listing shows a fitness scaler which reduces the the raw-fitness to its square root. This gives weaker individuals a greater changes being selected and weakens the influence of *super*-individuals.

When using a fitness scaler you have to take care that your scaler doesn't *destroy* your fitness value. This can be the case when your fitness value is negative and your fitness scaler squares the value. Trying to find the minimum will not work in this configuration.

3.3.3 Engine

The *evolution* `Engine` controls how the evolution steps are executed. Once the Engine is created, via a `Builder` class, it can't be changed. It doesn't contain any mutable global state and can therefore safely used/called from different threads. This allows to create more than one `EvolutionStreams` from the `Engine` and execute them in parallel.

```java
public final class Engine<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
    implements Function<EvolutionStart<G,C>, EvolutionResult<G,C>>
{
    // The evolution function, performs one evolution step.
    EvolutionResult<G,C> evolve(
        Population<G,C> population,
        long generation
    );

    // Evolution stream for "normal" evolution execution.
    EvolutionStream<G,C> stream();

    // Evolution iterator for external evolution iteration.
    Iterator<EvolutionResult<G,C>> iterator();
}
```

Listing 8: `Engine` class

Listing 8 shows the main methods of the `Engine` class. It is used for performing the actual evolution of a give population. One evolution step is executed by calling the `Engine.evolve` method, which returns an `EvolutionResult` object. This object contains the evolved `Population` plus additional information like execution duration of the several evolution sub-steps and information about the killed and as invalid marked individuals. With the `stream` method you create a new `EvolutionStream`, which is used for controlling the evolution process—see section 3.3.4 on page 24. Alternatively it is possible to iterate through the evolution process in an imperative way (for whatever reasons this should be necessary). Just create an `Iterator` of `EvolutionResult` object by calling the `iterator` method.

As already shown in previous examples, the `Engine` can only be created via its `Builder` class. Only the fitness `Function` and the `Chromosomes`, which represents the problem encoding, must be specified for creating an Engine instance.

For the rest of the parameters default values are specified. This are the `Engine` parameters which can configured:

`alterers` A list of `Alterer`s which are applied to the offspring `Population`, in the defined order. The default value of this property is set to `SinglePointCrossover<>(0.2)` followed by `Mutator<>(0.15)`.

`clock` The `java.time.Clock` used for calculating the execution durations. A `Clock` with nanosecond precision (`System.nanoTime()`) is used as default.

`executor` With this property it is possible to change the `java.util.concurrent.Executor` engine used for evaluating the evolution steps. This property can be used to define an application wide `Executor` or for controlling the number of execution threads. The default value is set to `ForkJoinPool.commonPool()`.

`fitnessFunction` This property defines the fitness `Function` used by the evolution `Engine`. (See section 3.3.1 on page 20.)

`fitnessScaler` This property defines the fitness scaler used by the evolution `Engine`. The default value is set to the identity function. (See section 3.3.2 on page 21.)

`genotypeFactory` Defines the `Genotype Factory` used for creating new individuals. Since the `Genotype` is its own `Factory`, it is sufficient to create a `Genotype`, which then serves as template.

`genotypeValidator` This property lets you *override* the default implementation of the `Genotype.isValid` method, which is useful if the `Genotype` validity not only depends on valid property of the elements it consists of.

`maximalPhenotypeAge` Set the maximal allowed age of an individual (Phenotype). This prevents *super* individuals to live *forever*. The default value is set to 70.

`offspringFraction` Through this property it is possible to define the fraction of offspring (and survivors) for evaluating the next generation. The fraction value must within the interval $[0, 1]$. The default value is set to 0.6. Additionally to this property, it is also possible to set the `survivorsFraction`, `survivorsSize` or `offspringSize`. All this additional properties effectively set the `offspringFraction`.

`offspringSelector` This property defines the `Selector` used for selecting the offspring `Population`. The default values is set to `TournamentSelector<>(3)`.

`optimize` With this property it is possible to define whether the fitness `Function` should be maximized of minimized. By default, the fitness `Function` is maximized.

`phenotypeValidator` This property lets you *override* the default implementation of the `Phenotype.isValid` method, which is useful if the `Phenotype` validity not only depends on valid property of the elements it consists of.

populationSize Defines the number of individuals of a `Population`. The evolution `Engine` keeps the number of individuals constant. That means, the `Population` of the `EvolutionResult` always contains the number of entries defined by this property. The default value is set to 50.

selector This method allows to set the `offspringSelector` and `survivorsSelector` in one step with the same selector.

survivorsSelector This property defines the `Selector` used for selecting the survivors `Population`. The default values is set to `TournamentSelector<>(3)`.

individualCreationRetries The evolution `Engine` tries to create only valid individuals. If a newly created `Genotype` is not valid, the `Engine` creates another one, till the created `Genotype` is valid. This parameter sets the maximal number of retries before the `Engine` gives up and accept invalid individuals. The default value is set to 10.

3.3.4 EvolutionStream

The `EvolutionStream` controls the execution of the evolution process and can be seen as a kind of execution *handle*. This handle can be used to define the termination criteria and to *collect* the final evolution result. Since the `EvolutionStream` extends the Java `Stream` interface, it integrates smoothly with the rest of the Java Stream API.[13]

```java
public interface EvolutionStream<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
    extends Stream<EvolutionResult<G, C>>
{
    public EvolutionStream<G, C>
    limit(Predicate<? super EvolutionResult<G, C>> proceed);
}
```

Listing 9: `EvolutionStream` class

Listing 9 shows the whole `EvolutionStream` interface. As it can be seen, it only adds one additional method. But this additional `limit` method allows to truncate the `EvolutionStream` based on a `Predicate` which takes an `EvolutionResult`. Once the `Predicate` returns `false`, the evolution process is stopped. Since the `limit` method returns an `EvolutionStream`, it is possible to define more than one `Predicate`, which both must be fulfilled to continue the evolution process.

```java
Engine<DobuleGene, Double> engine = ...
EvolutionStream<DoubleGene, Double> stream = engine.stream()
    .limit(predicate1)
    .limit(predicate2)
    .limit(100);
```

[13]It is recommended to make yourself familiar with the Java Stream API. A good introduction can be found here: `http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/`

The `EvolutionStream`, created in the example above, will be truncated if one of the two predicates is `false` or if the maximal allowed generations, of 100, is reached. An `EvolutionStream` is usually created via the `Engine.stream()` method. The *im*mutable and *stateless* nature of the evolution `Engine` allows to create more than one `EvolutionStream` with the same `Engine` instance.

> The generations of the `EvolutionStream` are evolved serially. Calls of the `EvolutionStream` methods (e. g. `limit`, `peek`, ...) are executed in the thread context of the created `Stream`. In a *typical* setup, no additional synchronization and/or locking is needed.

In cases where you appreciate the usage of the `EvolutionStream` but need a different `Engine` implementation, you can use the `EvolutionStream.of` factory method for creating an new `EvolutionStream`.

```java
static <G extends Gene<?, G>, C extends Comparable<? super C>>
EvolutionStream<G, C> of(
    Supplier<EvolutionStart<G, C>> start,
    Function<? super EvolutionStart<G, C>, EvolutionResult<G, C>> f
);
```

This factory method takes a start value, of type `EvolutionStart`, and an evolution `Function`. The evolution `Function` takes the start value and returns an `EvolutionResult` object. To make the runtime behavior more predictable, the start value is fetched/created lazily at the evolution start time.

```java
final Supplier<EvolutionStart<DoubleGene, Double>> start = ...
final EvolutionStream<DoubleGene, Double> stream =
    EvolutionStream.of(start, new MySpecialEngine());
```

3.3.5 EvolutionResult

The `EvolutionResult` collects the result data of an evolution step into an immutable *value* class. This class is the type of the stream elements of the EvolutionStream, as described in section 3.3.4 on the preceding page.

```java
public final class EvolutionResult<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
    implements Comparable<EvolutionResult<G, C>>
{
    Population<G,C> getPopulation();
    long getGeneration();
}
```

Listing 10: `EvolutionResult` class

Listing 3.3.5 shows the two most important properties, the `population` and the `generation` the result belongs to. This are also the two properties needed for the next evolution step. The `generation` is, of course, incremented by one. To make collecting the `EvolutionResult` object easier, it also implements the `Comparable` interface. Two `EvolutionResults` are compared by its best `Phenotype`.

The `EvolutionResult` classes has three predefined factory methods, which will return `Collectors` usable for the `EvolutionStream`:

`toBestEvolutionResult()` Collects the best `EvolutionResult` of an `EvolutionStream` according to the defined optimization strategy.

`toBestPhenotype()` This collector can be used if you are only interested in the best `Phenotype`.

`toBestGenotype()` Use this collector if you only need the best `Genotype` of the `EvolutionStream`.

The following code snippets shows how to use the different `EvolutionStream` collectors.

```
// Collecting the best EvolutionResult of the EvolutionStream.
EvolutionResult<DoubleGene, Double> result = stream
    .collect(EvolutionResult.toBestEvolutionResult());

// Collecting the best Phenotype of the EvolutionStream.
Phenotype<DoubleGene, Double> result = stream
    .collect(EvolutionResult.toBestPhenotype());

// Collecting the best Genotype of the EvolutionStream.
Genotype<DoubleGene> result = stream
    .collect(EvolutionResult.toBestGenotype());
```

3.3.6 EvolutionStatistics

The `EvolutionStatistics` class allows you to gather additional statistical information from the `EvolutionStream`. This is especially useful during the development phase of the application, when you have to find the right parametrization of the evolution `Engine`. Besides other informations, the `EvolutionStatistics` contains (statistical) information about the fitness, invalid and killed `Phenotypes` and runtime information of the different evolution steps. Since the `EvolutionStatistics` class implements the `Consumer<EvolutionResult<?, C>>` interface, it can be easily plugged into the `EvolutionStream`, adding it with the `peek` method of the stream.

```
Engine<DoubleGene, Double> engine = ...
EvolutionStatistics<?, Double> statistics =
    EvolutionStatistics.ofNumber();
engine.stream()
    .limit(100)
    .peek(statistics)
    .collect(toBestGenotype());
```

Listing 11: `EvolutionStatistics` usage

Listing 11 shows how to add the the `EvolutionStatistics` to the `EvolutionStream`. Once the algorithm tuning is finished, it can be removed in the production environment.

There are two different specializations of the `EvolutionStatistics` object available. The first is the general one, which will be working for every kind of `Genes` and fitness types. It can be created via the `EvolutionStatistics.ofComparable()` method. The second one collects additional statistical data for numeric fitness values. This can be created with the `EvolutionStatistics.ofNumber()` method.

```
+---------------------------------------------------------------------------+
|  Time statistics                                                          |
+---------------------------------------------------------------------------+
|             Selection: sum=0.046538278000 s; mean=0.003878189833 s        |
|              Altering: sum=0.086155457000 s; mean=0.007179621417 s        |
|   Fitness calculation: sum=0.022901606000 s; mean=0.001908467167 s        |
|     Overall execution: sum=0.147298067000 s; mean=0.012274838917 s        |
+---------------------------------------------------------------------------+
|  Evolution statistics                                                     |
+---------------------------------------------------------------------------+
|           Generations: 12                                                 |
|               Altered: sum=7,331; mean=610.916666667                      |
|                Killed: sum=0; mean=0.000000000                            |
|              Invalids: sum=0; mean=0.000000000                            |
+---------------------------------------------------------------------------+
|  Population statistics                                                    |
+---------------------------------------------------------------------------+
|                   Age: max=11; mean=1.951000; var=5.545190                |
|               Fitness:                                                    |
|                      min  = 0.000000000000                                |
|                      max  = 481.748227114537                              |
|                      mean = 384.430345078660                              |
|                      var  = 13006.132537301528                            |
+---------------------------------------------------------------------------+
```

A typical output of an number `EvolutionStatistics` object will look like the example above.

The `EvolutionStatistics` object is a simple for inspecting the `EvolutionStream` after it is finished. It doesn't give you a *live* view of the current evolution process, which can be necessary for long running streams. In such cases you have to maintain/update the statistics yourself.

```java
public class TSM {
    // The locations to visit.
    static final ISeq<Point> POINTS = ISeq.of(...);

    // The permutation codec.
    static final Codec<ISeq<Point>, EnumGene<Point>>
    CODEC = codecs.ofPermutation(POINTS);

    // The fitness function (in the problem domain).
    static double dist(final ISeq<Point> p) {...}

    // The evolution engine.
    static final Engine<EnumGene<Point>, Double> ENGINE = Engine
        .builder(TSM::dist, CODEC)
        .optimize(Optimize.MINIMUM)
        .build();

    // Best phenotype found so far.
    static Phenotype<EnumGene<Point>, Double> best = null;

    // You will be informed on new results. This allows to
    // react on new best phenotypes, e.g. log it.
    private static void update(
        final EvolutionResult<EnumGene<Point>, Double> result
    ) {
        if (best == null ||
            best.compareTo(result.getBestPhenotype()) < 0)
        {
            best = result.getBestPhenotype();
            System.out.print(result.getGeneration() + ": ");
            System.out.println("Found best phenotype: " + best);
        }
    }

```

```java
    // Find the solution.
    public static void main(final String[] args) {
        final ISeq<Point> result = CODEC.decode(
            ENGINE.stream()
                .peek(TSM::update)
                .limit(10)
                .collect(EvolutionResult.toBestGenotype())
        );
        System.out.println(result);
    }
}
```

Listing 12: Live evolution statistics

Listing 12 on the previous page shows how to implement a manual statistics gathering. The update method is called whenever a new `EvolutionResult` is has been calculated. If a new best `Phenotype` is available, it is stored and logged. With the `TSM::update` method, which is called on every finished generation, you have a *live* view on the evolution progress.

4 Nuts and bolts

4.1 Concurrency

The **Jenetics** library parallelizes independent task whenever possible. Especially the evaluation of the fitness function is done concurrently. That means that the fitness function must be thread safe, because it is shared by all phenotypes of a population. The easiest way for achieving thread safety is to make the fitness function immutable and re-entrant.

4.1.1 Basic configuration

The used `Executor` can be defined when building the evolution `Engine` object.

```java
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class Main {
    private static Double eval(final Genotype<DoubleGene> gt) {
        // calculate and return fitness
    }

    public static void main(final String[] args) {
        // Creating an fixed size ExecutorService
        final ExecutorService executor = Executors
            .newFixedThreadPool(10)
        final Factory<Genotype<DoubleGene>> gtf = ...
        final Engine<DoubleGene, Double> engine = Engine
            .builder(Main::eval, gtf)
             // Using 10 threads for evolving.
            .executor(executor)
            .build()
        ...
    }
}
```

If no `Executor` is given, **Jenetics** uses a common `ForkJoinPool`[14] for concur-

[14] `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html`

rency.

Sometimes it might be useful to run the evaluation `Engine` single-threaded, or even execute all operations in the main thread. This can be easily achieved by setting the appropriate `Executor`.

```java
final Engine<DoubleGene, Double> engine = Engine.builder(...)
    // Doing the Engine operations in the main thread
    .executor((Executor)Runnable::run)
    .build()
```

The code snippet above shows how to do the `Engine` operations in the main thread. Whereas the snippet below executes the `Engine` operations in a single thread, other than the main thread.

```java
final Engine<DoubleGene, Double> engine = Engine.builder(...)
    // Doing the Engine operations in a single thread
    .executor(Executors.newSingleThreadExecutor())
    .build()
```

Such a configuration can be useful for performing reproducible (performance) tests, without the uncertainty of a concurrent execution environment.

4.1.2 Concurrency tweaks

Jenetics uses different strategies for minimizing the concurrency overhead, depending on the configured `Executor`. For the `ForkJoinPool`, the fitness evaluation of the population is done by recursively dividing it into sub-populations using the abstract `RecursiveAction` class. If a minimal sub-population size is reached, the fitness values for this sub-population are directly evaluated. The default value of this threshold is five and can be controlled via the `io.jenetics.concurrency.splitThreshold` system property. Besides the `splitThreshold`, the size of the evaluated sub-population is dynamically determined by the `ForkJoinTask.getSurplusQueuedTaskCount()` method.[15] If this value is greater than three, the fitness values of the current sub-population are also evaluated immediately. The default value can be overridden by the `io.jenetics.concurrency.maxSurplusQueuedTaskCount` system property.

```
$ java -Dio.jenetics.concurrency.splitThreshold=1 \
    -Dio.jenetics.concurrency.maxSurplusQueuedTaskCount=2 \
    -cp jenetics-3.8.0.jar:app.jar \
    com.foo.bar.MyJeneticsApp
```

You may want to tweak this parameters, if you realize a low CPU utilization during the fitness value evaluation. Long running fitness function could lead to CPU under-utilization while evaluating the last sub-population. In this case, only one core is busy, while the other cores are idle, because they already finished the fitness evaluation. Since the workload has been already distributed, no *work-stealing* is possible. Reducing the `splitThreshold` can help to have a more equal workload distribution between the available CPUs. Reducing the

[15] Excerpt from the Javadoc: *Returns an estimate of how many more locally queued tasks are held by the current worker thread than there are other worker threads that might steal them. This value may be useful for heuristic decisions about whether to fork other tasks. In many usages of `ForkJoinTasks`, at steady state, each worker should aim to maintain a small constant surplus (for example, 3) of tasks, and to process computations locally if this threshold is exceeded.*

maxSurplusQueuedTaskCount property will create a more uniform workload for fitness function with heavily varying computation cost for different genotype values.

The fitness function shouldn't acquire locks for achieving thread safety. It is also recommended to avoid calls to blocking methods. If such calls are unavoidable, consider using the ForkJoinPool.managedBlock method. Especially if you are using a ForkJoinPool executor, which is the default.

If the Engine is using an **ExecutorService**, a different optimization strategy is used for reducing the concurrency overhead. The original population is divided into a fixed number[16] of sub-populations, and the fitness values of each sub-population are evaluated by one thread. For long running fitness functions, it is better to have smaller sub-populations for a better CPU utilization. With the io.jenetics.concurrency.maxBatchSize system property, it is possible to reduce the sub-population size. The default value is set to Integer.MAX_VALUE. This means, that only the number of CPU cores influences the *batch* size.

```
$ java -Dio.jenetics.concurrency.maxBatchSize=3 \
    -cp jenetics-3.8.0.jar:app.jar \
    com.foo.bar.MyJeneticsApp
```

Another source of under-utilized CPUs are lock contentions. It is therefore strongly recommended to avoid locking and blocking calls in your fitness function at all. If blocking calls are unavoidable, consider using the *managed block* functionality of the ForkJoinPool.[17]

4.2 Randomness

In general, GAs heavily depends on *pseudo* random number generators (PRNG) for creating new individuals and for the selection- and mutation-algorithms. **Jenetics** uses the Java Random object, respectively sub-types from it, for generating random numbers. To make the random engine pluggable, the Random object is always fetched from the RandomRegistry. This makes it possible to change the implementation of the random engine without changing the client code. The central RandomRegistry also allows to easily change Random engine even for specific parts of the code.

The following example shows how to change and restore the Random object. When opening the with scope, changes to the RandomRegistry are only visible within this scope. Once the with scope is left, the original Random object is restored.

```java
List<Genotype<DoubleGene>> genotypes =
    RandomRegistry.with(new Random(123), r -> {
        Genotype.of(DoubleChromosome.of(0.0, 100.0, 10))
```

[16]The number of sub-populations actually depends on the number of available CPU cores, which are determined with Runtime.availableProcessors().

[17]A good introduction on how to use managed blocks, and the motivation behind it, is given in this talk: https://www.youtube.com/watch?v=rUDGQQ83ZtI

```java
            .instances()
            .limit(100)
            .collect(Collectors.toList())
    });
```

With the previous listing, a random, but reproducible, list of genotypes is created. This might be useful while testing your application or when you want to evaluate the `EvolutionStream` several times with the same initial population.

```java
Engine<DoubleGene, Double> engine = ...;
// Create a new evolution stream with the given
// initial genotypes.
Phenotype<DoubleGene, Double> best = engine.stream(genotypes)
    .limit(10)
    .collect(EvolutionResult.toBestPhenotype());
```

The example above uses the generated genotypes for creating the `EvolutionStream`. Each created stream uses the same starting population, but will, most likely, create a different result. This is because the stream evaluation is still non-deterministic.

Setting the PRNG to a Random object with a defined seed has the effect, that every evolution *stream* will produce the same result—in an single threaded environment.

The parallel nature of the GA implementation requires the creation of streams $t_{i,j}$ of random numbers which are statistically independent, where the streams are numbered with $j = 1, 2, 3, ..., p$, p denotes the number of processes. We expect statistical independence between the streams as well. The used PRNG should enable the GA to *play fair*, which means that the outcome of the GA is strictly independent from the underlying hardware and the number of parallel processes or threads. This is essential for reproducing results in parallel environments where the number of parallel tasks may vary from run to run.

The *Fair Play* property of a PRNG guarantees that the quality of the genetic algorithm (evolution stream) does not depend on the degree of parallelization.

When the `Random` engine is used in an multi-threaded environment, there must be a way to parallelize the sequential PRNG. Usually this is done by taking the elements of the sequence of pseudo-random numbers and distribute them among the threads. There are essentially four different parallelizations techniques used in practice: *Random seeding*, *Parameterization*, *Block splitting* and *Leapfrogging*.

Random seeding Every thread uses the same kind of PRNG but with a different seed. This is the default strategy used by the **Jenetics** library. The `RandomRegistry` is initialized with the `ThreadLocalRandom` class from the `java.util.concurrent` package. Random seeding works well for the most problems but without theoretical foundation.[18] If you assume that this strategy is responsible for some *non*-reproducible results, consider using the `LCG64ShiftRandom` PRNG instead, which uses *block splitting* as parallelization strategy.

Parameterization All threads uses the same kind of PRNG but with different parameters. This requires the PRNG to be parameterizable, which is not the case for the Random object of the JDK. You can use the `LCG64ShiftRandom` class if you want to use this strategy. The theoretical foundation for these method is weak. In a massive parallel environment you will need a reliable set of parameters for every random stream, which are not trivial to find.

Block splitting With this method each thread will be assigned a non-overlapping contiguous block of random numbers, which should be enough for the whole runtime of the process. If the number of threads is not known in advance, the length of each block should be chosen much larger then the maximal expected number of threads. This strategy is used when using the `LCG64ShiftRandom.ThreadLocal` class. This class assigns every thread a block of $2^{56} \approx 7,2 \cdot 10^{16}$ random numbers. After 128 threads, the blocks are recycled, but with changed seed.

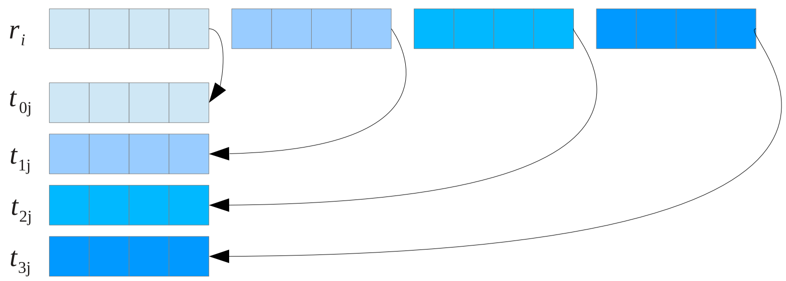


Figure 4.1: Block splitting

Leapfrog With the leapfrog method each thread $t \in [0, P)$ only consumes the P^{th} random number and jump ahead in the random sequence by the number of threads, P. This method requires the ability to jump very quickly ahead in the sequence of random numbers by a given amount. Figure 4.2 on the following page graphically shows the concept of the *leapfrog* method.

`LCG64ShiftRandom` The `LCG64ShiftRandom` class is a port of the `trng::lcg64_shift` PRNG class of the TRNG[19] library, implemented in C++.[4] It implements additional methods, which allows to implement the *block splitting*—and also the *leapfrog*—method.

```java
public class LCG64ShiftRandom extends Random {
    public void split(final int p, final int s);
    public void jump(final long step);
    public void jump2(final int s);
```

[18] This is also expressed by Donald Knuth's advice: »Random number generators should not be chosen at random.«

[19] `http://numbercrunch.de/trng/`

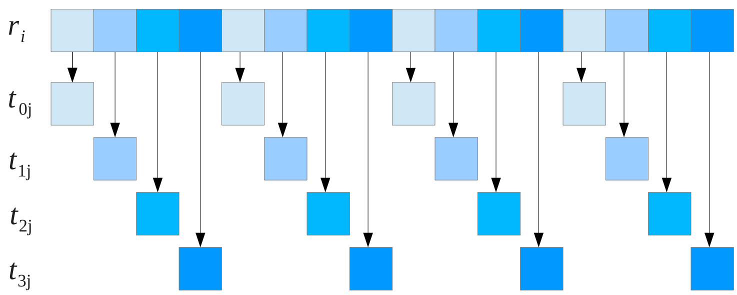


Figure 4.2: Leapfrogging

```
    ...
}
```

Listing 13: `LCG64ShiftRandom` class

Listing 13 shows the interface used for implementing the block splitting and leapfrog parallelizations technique. This methods have the following meaning:

`split` Changes the internal state of the PRNG in a way that future calls to `nextLong()` will generated the s^{th} sub-stream of p^{th} sub-streams. s must be within the range of $[0, p-1)$. This method is used for parallelization via *leapfrogging*.

`jump` Changes the internal state of the PRNG in such a way that the engine jumps$s$ steps ahead. This method is used for parallelization via *block splitting*.

`jump2` Changes the internal state of the PRNG in such a way that the engine jumps 2^s steps ahead. This method is used for parallelization via *block splitting*.

Runtime performance Table 4.1 shows the random number generation speed for the different PRNG implementations.[20]

	`int`/s	`long`/s	`float`/s	`double`/s
`Random`	$87 \cdot 10^6$	$43 \cdot 10^6$	$86 \cdot 10^6$	$42 \cdot 10^6$
`ThreadLocalRandom`	$255 \cdot 10^6$	$253 \cdot 10^6$	$208 \cdot 10^6$	$208 \cdot 10^6$
`LCG64ShiftRandom`	$237 \cdot 10^6$	$241 \cdot 10^6$	$176 \cdot 10^6$	$178 \cdot 10^6$

Table 4.1: Performance of various PRNG implementations.

The default PRNG used by the **Jenetics** has the best runtime performance behavior (for generating `int` values).

[20]Measured on a Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz with Java(TM) SE Runtime Environment (build 1.8.0_102-b14)—Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)—, using the JHM micro-benchmark library.

4.3 Serialization

Jenetics supports serialization for a number of classes, most of them are located in the `org.jenetics` package. Only the concrete implementations of the `Gene` and the `Chromosome` interfaces implements the `Serializable` interface. This gives a greater flexibility when implementing own `Genes` and `Chromosomes`.

- `BitGene`
- `BitChromosome`
- `CharacterGene`
- `CharacterChromosome`
- `IntegerGene`
- `IntegerChromosome`
- `LongGene`
- `LongChomosome`
- `DoubleGene`
- `DoubleChromosome`
- `EnumGene`
- `PermutationChromosome`
- `Genotype`
- `Phenotype`
- `Population`

With the serialization mechanism you can write a population to disk and load it into an new `EvolutionStream` at a later time. It can also be used to transfer populations to evolution engines, running on different hosts, over a network link. The `IO` class, located in the `org.jenetics.util` package, supports native Java serialization and JAXB XML serialization.

```java
// Creating result population.
EvolutionResult<DoubleGene, Double> result = stream
    .limit(100)
    .collect(toBestEvolutionResult());

// Writing the population to disk.
final File file = new File("population.xml");
IO.jaxb.write(result.getPopulation(), file);

// Reading the population from disk.
Population<DoubleGene, Double> population =
    (Population<DoubleGene, Double>)IO.jaxb.read(file);
EvolutionStream<DoubleGene, Double> stream = Engine
    .build(ff, gtf)
    .stream(population, 1);
```

The following listing shows the XML serialization of a `Population` which consists of `Genotypes` as shown in figure 3.2 on page 7; only the first `Phenotype` is shown.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<org.jenetics.Population size="5">
    <phenotype
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="org.jenetics.Phenotype" generation="294"
    >
        <genotype length="5" ngenes="100">
            <chromosome
                xsi:type="org.jenetics.DoubleChromosome"
                length="20" min="0.0" max="1.0"
            >
                <allele>0.27251556008507416</allele>
                <allele>0.003140816229067145</allele>
```

```
                <allele>0.43947528327497376</allele>
                <allele>0.10654807463069327</allele>
                <allele>0.19696530915810317</allele>
                <allele>0.7450003838065538</allele>
                <allele>0.5594416969271359</allele>
                <allele>0.02823782430152355</allele>
                <allele>0.5741102315010789</allele>
                <allele>0.4533651041367144</allele>
                <allele>0.811148141800367</allele>
                <allele>0.5710456351848858</allele>
                <allele>0.30166768355230955</allele>
                <allele>0.5455492865240272</allele>
                <allele>0.21068427527733102</allele>
                <allele>0.5265067943902246</allele>
                <allele>0.273549098065591</allele>
                <allele>0.2648197379297126</allele>
                <allele>0.8732775776362911</allele>
                <allele>0.9498003919007005</allele>
            </chromosome>
            ...
        </genotype>
        <fitness
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xsi:type="xs:double"
        >234.23443</fitness>
        <raw-fitness
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xsi:type="xs:double"
        >34.2498</raw-fitness>
    </phenotype>
    ...
</org.jenetics.Population>
```

When serializing a whole population the fitness function and fitness scaler are not part of the serialized XML file. If an `EvolutionStream` is initialized with a previously serialized `Population`, the `Engine`'s current fitness function and fitness scaler are used for *re*-calculating the fitness values.

The `IO` class can also be used for serializing own JAXB annotated classes. Listing 14 shows how an user-defined JAXB class can be marshaled with the `IO` helper class.

```java
@XmlRootElement(name = "data-class")
@XmlType(name = "DataClass")
@XmlAccessorType(XmlAccessType.FIELD)
public class DataClass {
    @XmlAttribute public String name;
    @XmlValue public String value;

    public DataClass(final String name, final String value) {
        this.name = name;
        this.value = value;
    }

    // Default constructor needed by JAXB.
    public DataClass() {
    }

    public static void main(final String[] args) throws Exception {
        // Registering the class before serialization.
        IO.JAXB.register(DataClass.class);

        final DataClass data =
            new DataClass("some name", "some value");
        IO.jaxb.write(data, System.out);
    }
```

```
}
```

Listing 14: *DataClass* JAXB serialization

The output of the marshaled `DataClass` looks like expected.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<data-class name="some name">some value</data-class>
```

4.4 Utility classes

The `org.jenetics.util` and the `org.jenetics.stat` package of the library contains utility and helper classes which are essential for the implementation of the GA.

`org.jenetics.util.Seq` Most notable are the `Seq` interfaces and its implementation. They are used, among others, in the `Chromosome` and `Genotype` classes and holds the `Genes` and `Chromosomes`, respectively. The `Seq` interface itself represents a fixed-sized, ordered sequence of elements. It is an abstraction over the Java build-in *array*-type, but much safer to use for *generic* elements, because there are no casts needed when using *nested* generic types.

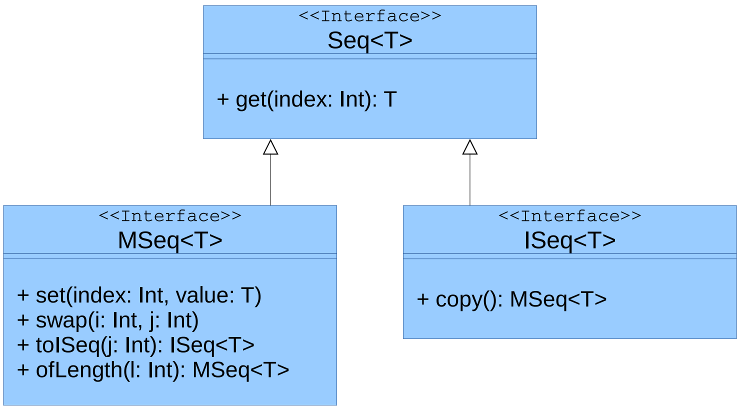


Figure 4.3: `Seq` class diagram

Figure 4.3 shows the `Seq` class diagram with their most important methods. The interfaces `MSeq` and `ISeq` are mutable, respectively immutable specializations of the basis interface. Creating instances of the `Seq` interfaces is possible via the static factory methods of the interfaces.

```java
// Create "different" sequences.
final Seq<Integer> a1 = Seq.of(1, 2, 3);
final MSeq<Integer> a2 = MSeq.of(1, 2, 3);
final ISeq<Integer> a3 = MSeq.of(1, 2, 3).toISeq();
final MSeq<Integer> a4 = a3.copy();

// The 'equals' method performs element-wise comparison.
assert(a1.equals(a2) && a1 != a2);
assert(a2.equals(a3) && a2 != a3);
assert(a3.equals(a4) && a3 != a4);
```

How to create instances of the three `Seq` types is shown in the listing above. The `Seq` classes also allows a more *functional* programming style. For a full method description refer to the Javadoc.

`org.jenetics.stat` This package contains classes for calculating statistical moments. They are designed to work smoothly with the Java Stream API and are divided into mutable (number) consumers and immutable value classes, which holds the statistical moments. The additional classes calculate the

- *minimum*,
- *maximum*,
- *sum*,
- *mean*,
- *variance*,
- *skewness* and
- *kurtosis* value.

Numeric type	Consumer class	Value class
`int`	`IntMomentStatistics`	`IntMoments`
`long`	`LongMomentStatistics`	`LongMoments`
`double`	`DoubleMomentStatistics`	`DoubleMoments`

Table 4.2: Statistics classes

Table 4.2 contains the available statistical moments for the different numeric types. The following code snippet shows an example on how to collect double statistics from an given `DoubleGene` stream.

```java
// Collecting into an statistics object.
DoubleChromosome chromosome = ...
DoubleMomentStatistics statistics = chromosome.stream()
    .collect(DoubleMomentStatistics
        .toDoubleMomentStatistics(v -> v.doubleValue()));

// Collecting into an moments object.
DoubleMoments moments = chromosome.stream()
    .collect(DoubleMoments.toDoubleMoments(v -> v.doubleValue()));
```

5 Extending **Jenetics**

The **Jenetics** library was designed to give you a great flexibility in transforming your problem into a structure that can be solved by an GA. It also comes with different implementations for the base data-types (genes and chromosomes) and operators (alterers and selectors). If it is still some functionality missing, this section describes how you can extend the existing classes. Most of the *extensible* classes are defined by an interface and have an abstract implementation which makes it easier to extend it.

5.1 Genes

`Gene`s are the starting point in the class hierarchy. They hold the actual information, the alleles, of the problem domain. Beside the *classical* bit-gene, **Jenetics** comes with gene implementations for numbers (double-, int- and long values), characters and enumeration types.

For implementing your own gene type you have to implement the Gene interface with three methods: (1) the getAllele() method which will return the wrapped data, (2) the newInstance method for creating new, random instances of the gene—must be of the same type and have the same constraint—and (3) the isValid() method which checks if the gene fulfill the expected constraints. The gene constraint might be violated after mutation and/or recombination. If you want to implement a new number-gene, e. g. a gene which holds complex values, you may want extend it from the abstract NumericGene class. Every Gene extends the Serializable interface. For *normal* genes there is no more work to do for using the Java serialization mechanism.

The custom Genes and Chromosomes implementations must use the Random engine available via the RandomRegistry.getRandom method when implementing their factory methods. Otherwise it is not possible to seamlessly change the Random engine by using the RandomRegistry.setRandom method.

If you want to support your own allele type, but want to avoid the effort of implementing the Gene interface, you can alternatively use the AnyGene class. It can be created with AnyGene.of(Supplier, Predicate). The given Supplier is responsible for creating new random alleles, similar to the newInstance method in the Gene interface. Additional validity checks are performed by the given Predicate.

```java
class LastMonday {
    // Creates new random 'LocalDate' objects.
    private static LocalDate nextMonday() {
        final Random random = RandomRegistry.getRandom();
        LocalDate
            .of(2015, 1, 5)
            .plusWeeks(random.nextInt(1000));
    }

    // Do some additional validity check.
    private static boolean isValid(final LocalDate date) {...}

    // Create a new gene from the random 'Supplier' and
    // validation 'Predicate'.
    private final AnyGene<LocalDate> gene = AnyGene
        .of(LastMonday::nextMonday, LastMonday::isValid);
}
```

Listing 15: AnyGene example

Example listing 15 shows the (almost) minimal setup for creating user defined Gene allele types. By convention, the Random engine, used for creating the new LocalDate objects, must be requested from the RandomRegistry. With the optional validation function, isValid, it is possible to reject Genes whose alleles doesn't conform some criteria.

The simple usage of the AnyGene has also its downsides. Since the AnyGene instances are created from function objects, serialization is not supported by the `AnyGene` class. It is also not possible to use some `Alterer` implementations with the `AnyGene`, like:

- `GaussianMutator`,
- `MeanAlterer` and
- `PartiallyMatchedCrossover`

5.2 Chromosomes

A new gene type normally needs a corresponding chromosome implementation. The most important part of a chromosome is the factory method `newInstance`, which lets the evolution `Engine` create a new `Chromosome` instance from a sequence of `Genes`. This method is used by the `Alterers` when creating new, combined `Chromosomes`. It is allowed, that the newly created chromosome has a different length than the original one. The other methods should be self-explanatory. The chromosome has the same serialization mechanism as the gene. For the minimal case it can extends the `Serializable` interface.

Corresponding to the `AnyGene`, it is possible to create chromosomes with arbitrary allele types with the `AnyChromosome`.

```java
public class LastMonday {
    // The used problem Codec.
    private static final Codec<LocalDate, AnyGene<LocalDate>>
    CODEC = Codec.of(
        Genotype.of(AnyChromosome.of(LastMonday::nextMonday)),
        gt -> gt.getGene().getAllele()
    );

    // Creates new random 'LocalDate' objects.
    private static LocalDate nextMonday() {
        final Random random = RandomRegistry.getRandom();
        LocalDate
            .of(2015, 1, 5)
            .plusWeeks(random.nextInt(1000));
    }

    // The fitness function: find a monday at the end of the month.
    private static int fitness(final LocalDate date) {
        return date.getDayOfMonth();
    }

    public static void main(final String[] args) {
        final Engine<AnyGene<LocalDate>, Integer> engine = Engine
            .builder(LastMonday::fitness, CODEC)
            .offspringSelector(new RouletteWheelSelector<>())
            .build();

        final Phenotype<AnyGene<LocalDate>, Integer> best =
            engine.stream()
                .limit(50)
                .collect(EvolutionResult.toBestPhenotype());

        System.out.println(best);
    }
}
```

Listing 16: `AnyChromosome` example

Listing 16 on the preceding page shows a full usage example of the `AnyGene` and `AnyChromosome` class. The example tries to find a Monday with a maximal day of month. An interesting detail is, that an `Codec`[21] definition is used for creating new `Genotypes` and for converting them back to `LocalDate` alleles.

The convenient usage of the `AnyChromosome` has to be payed by the same restriction as for the `AnyGene`: no serialization support for the chromosome and not usable for all `Alterer` implementations.

5.3 Selectors

If you want to implement your own selection strategy you only have to implement the `Selector` interface with the `select` method.

```java
@FunctionalInterface
public interface Selector<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
> {
    public Population<G, C> select(
        Population<G, C> population,
        int count,
        Optimize opt
    );
}
```

Listing 17: `Selector` interface

The first parameter is the original `population` from which the *sub*-population is selected. The second parameter, `count`, is the number of individuals of the returned sub-population. Depending on the selection algorithm, it is possible that the sub-population contains more elements than the original one. The last parameter, `opt`, determines the optimization strategy which must be used by the selector. This is exactly the point where it is decided whether the GA minimizes or maximizes the fitness function.

Before implementing a selector from scratch, consider to extend your selector from the `ProbabilitySelector` (or any other available `Selector` implementation). It is worth the effort to try to express your selection strategy in terms of selection property $P(i)$. Another way for re-using existing `Selector` implementation is by composition.

```java
public class EliticSelector<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
>
    implements Selector<G, C>
{
    private final TruncationSelector<G, C>
    _elite = new TruncationSelector<>();

    private final TournamentSelector<G, C>
    _rest = new TournamentSelector<>(3);

    public EliticSelector() {
    }

    @Override
```

[21] See section 6.2 on page 49 for a more detailed `Codec` description.

```
    public Population<G, C> select(
        final Population<G, C> population,
        final int count,
        final Optimize opt
    ) {
        return population.isEmpty() || count <= 0
            ? new Population<>(0)
            : append(
                _elite.select(population, 1, opt),
                _rest.select(population, max(0, count - 1), opt));
    }

    private Population<G, C> append(
        final Population<G, C> p1,
        final Population<G, C> p2
    ) {
        p1.addAll(p2);
        return p1;
    }
}
```

Listing 18: Elitist selector

Listing 18 on the preceding page shows how an *elitist* selector could be implemented by using the existing `Truncation`- and `TournamentSelector`. With *elitist* selection, the quality of the best solution in each generation monotonically increases over time.[3] Although this is not necessary, since the evolution `Engine`/`Stream` doesn't throw away the best solution found during the evolution process.

5.4 Alterers

For implementing a new alterer class it is necessary to implement the `Alterer` interface. You might do this if your new `Gene` type needs a special kind of alterer not available in the **Jenetics** project.

```
@FunctionalInterface
public interface Alterer<
    G extends Gene<?, G>,
    C extends Comparable<? super C>
> {
    public int alter(
        Population<G, C> population,
        long generation
    );
}
```

Listing 19: `Alterer` interface

The first parameter of the `alter` method is the `Population` which has to be altered. Since the the `Population` class is mutable, the altering is performed in place. The second parameter is the `generation` of the newly created individuals and the return value is the number of genes that has been altered.

To maximize the range of application of an `Alterer`, it is recommended that they can handle `Genotype`s and `Chromosome`s with variable length.

5.5 Statistics

During the developing phase of an application which uses the **Jenetics** library, additional statistical data about the evolution process is crucial. Such data can help to optimize the parametrization of the evolution Engine. A good starting point is to use the `EvolutionStatistics` class in the `org.jenetics.engine` package (see listing 11 on page 26). If the data in the `EvolutionStatistics` class doesn't fit your needs, you simply have to write your own statistics class. It is not possible to derive from the existing `EvolutionStatistics` class. This is not a real restriction, since you still can use the class by delegation. Just implement the Java `Consumer<EvolutionResult<G, C>>` interface.

5.6 Engine

The evolution `Engine` itself can't be extended, but it is still possible to create an `EvolutionStream` without using the `Engine` class.[22] Because the `EvolutionStream` has no direct dependency to the `Engine`, it is possible to use an different, special evolution `Function`.

```java
public final class SpecialEngine {
    // The Genotype factory.
    private static final Factory<Genotype<DoubleGene>> GTF =
        Genotype.of(DoubleChromosome.of(0, 1));

    // The fitness function.
    private static Double fitness(final Genotype<DoubleGene> gt) {
        return gt.getGene().getAllele();
    }

    // Create new evolution start object.
    private static EvolutionStart<DoubleGene, Double>
    start(final int populationSize, final long generation) {
        final Population<DoubleGene, Double> population = GTF
            .instances()
            .map(gt -> Phenotype
                .of(gt, generation, SpecialEngine::fitness))
            .limit(populationSize)
            .collect(Population.toPopulation());

        return EvolutionStart.of(population, generation);
    }

    // The special evolution function.
    private static EvolutionResult<DoubleGene, Double>
    evolve(final EvolutionStart<DoubleGene, Double> start) {
        return ...; // Add implementation!
    }

    public static void main(final String[] args) {
        final Genotype<DoubleGene> best = EvolutionStream
            .of(() -> start(50, 0), SpecialEngine::evolve)
            .limit(limit.bySteadyFitness(10))
            .limit(100)
            .collect(EvolutionResult.toBestGenotype());

        System.out.println("Best Genotype: " + best));
```

[22] Also refer to section 3.3.4 on page 24 on how to create an `EvolutionStream` from an evolution `Function`.

```
    }
}
```

Listing 20: Special evolution engine

Listing 20 on the previous page shows a *complete* implementation stub for using an own special evolution `Function`.

6 Advanced topics

This section describes some advanced topics for setting up an evolution `Engine` or `EvolutionStream`. It contains some problem encoding examples and how to override the default validation strategy of the given Genotypes. The last section contains a detailed description of the implemented termination strategies.

6.1 Encoding

This section presents some encoding examples for common problems. The encoding should be a complete and minimal expression of a solution to the problem. An encoding is complete if it contains enough information to represent every solution to the problem. An minimal encoding contains only the information needed to represent a solution to the problem. If an encoding contains more information than is needed to uniquely identify solutions to the problem, the search space will be larger than necessary.

Whenever possible, the encoding should not be able to represent infeasible solutions. If a genotype can represent an infeasible solution, care must be taken in the fitness function to give partial credit to the genotype for its »good« genetic material while sufficiently penalizing it for being infeasible. Implementing a specialized `Chromosome`, which won't create invalid encodings can be a solution to this problem. In general, it is much more desirable to design a representation that can only represent valid solutions so that the fitness function measures only fitness, not validity. An encoding that includes invalid individuals enlarges the search space and makes the search more costly. A deeper analysis of how to create encodings can be found in [15] and [14].

Some of the encodings represented in the following sections has been implemented by **Jenetics**, using the `Codec`[23] interface, and are available through static factory methods of the `org.jenetics.engine.codecs` class.

6.1.1 Real function

Jenetics contains three different numeric gene and chromosome implementations, which can be used to encode a real function, $f : \mathbb{R} \to \mathbb{R}$:

- `IntegerGene/Chromosome`,
- `LongGene/Chromosome` and
- `DoubleGene/Chromosome`.

It is quite easy to encode a real function. Only the minimum and maximum value of the function domain must be defined. The `DoubleChromosome` of length 1 is then wrapped into a `Genotype`.

[23]See section 6.2 on page 49.

```java
Genotype.of(
    DoubleChromosome.of(min, max, 1)
);
```

Decoding the double value from the Genotype is also straight forward. Just get the first gene from the first chromosome, with the `getGene()` method, and convert it to a `double`.

```java
static double toDouble(final Genotype<DoubleGene> gt) {
    return gt.getGene().doubleValue();
}
```

When the `Genotype` only contains *scalar* chromosomes[24], it should be clear, that it can't be altered by every `Alterer`. That means, that none of the `Crossover` alterers will be able to create modified `Genotypes`. For *scalars* the appropriate alterers would be the `MeanAlterer`, `GaussianAlterer` and `Mutator`.

Scalar `Chromosomes` and/or `Genotypes` can only be altered by `MeanAlterer`, `GaussianAlterer` and `Mutator` classes. Other alterers are allowed, but will have no effect on the `Chromosomes`.

6.1.2 Scalar function

Optimizing a function $f(x_1, ..., x_n)$ of one or more variable whose range is one-dimensional, we have two possibilities for the Genotype encoding.[19] For the *first* encoding we expect that all variables, x_i, have the same minimum and maximum value. In this case we can simply create a `Genotype` with a *`Numeric`*`Chromosome` of the desired length n.

```java
Genotype.of(
    DoubleChromosome.of(min, max, n)
);
```

The decoding of the `Genotype` requires a cast of the first `Chromosome` to a `DoubleChromosome`. With a call to the `DoubleChromosome.toArray()` method we return the variables $(x_1, ..., x_n)$ as `double[]` array.

```java
static double[] toScalars(final Genotype<DoubleGene> gt) {
    return gt.getChromosome().as(DoubleChromosome.class).toArray();
}
```

With the *first* encoding you have the possibility to use all available alterers, including all `Crossover` alterer classes.

The *second* encoding *must* be used if the minimum and maximum value of the variables x_i can't be the same for all i. For the different domains, each variable x_i is represented by a *`Numeric`*`Chromosome` with length one. The final `Genotype` will consist of n `Chromosomes` with length one.

```java
Genotype.of(
    DoubleChromosome.of(min1, max1, 1),
    DoubleChromosome.of(min2, max2, 1),
    ...
    DoubleChromosome.of(minn, maxn, 1)
);
```

[24]Scalar chromosomes contains only one gene.

With the help of the new Java Stream API, the decoding of the `Genotype` can be done in a view lines. The `DoubleChromosome` stream, which is created from the chromosome `Seq`, is first mapped to `double` values and then collected into an array.

```java
static double[] toScalars(final Genotype<DoubleGene> gt) {
    return gt.stream()
        .mapToDouble(c -> c.getGene().doubleValue())
        .toArray();
}
```

As already mentioned, with the use of scalar chromosomes we can only use the `MeanAlterer`, `GaussianAlterer` or `Mutator` alterer class.

If there are performance issues in converting the `Genotype` into a `double[]` array, or any other numeric array, you can access the `Genes` directly via the `Genotype.get(i, j)` method and than convert it to the desired numeric value, by calling `intValue()`, `longValue()` or `doubleValue()`.

6.1.3 Vector function

A function $f(X_1, ..., X_n)$, of one to n variables whose range is m-dimensional, is encoded by m `DoubleChromosomes` of length n.[20] The domain–minimum and maximum values–of one variable X_i are the same in this encoding.

```java
Genotype.of(
    DoubleChromosome.of(min1, max1, m),
    DoubleChromosome.of(min2, max2, m),
    ...
    DoubleChromosome.of(minn, maxn, m)
);
```

The decoding of the vectors is quite easy with the help of the Java Stream API. In the first `map` we have to cast the `Chromosome<DoubleGene>` object to the actual `DoubleChromosome`. The second `map` then converts each `DoubleChromosome` to an `double[]` array, which is collected to an 2-dimensional `double[n][m]` array afterwards.

```java
static double[][] toVectors(final Genotype<DoubleGene> gt) {
    return gt.stream()
        .map(dc -> dc.as(DoubleChromosome.class).toArray())
        .toArray(double[][]::new);
}
```

For the special case of $n = 1$, the decoding of the `Genotype` can be simplified to the decoding we introduced for scalar functions in section 6.1.2.

```java
static double[] toVector(final Genotype<DoubleGene> gt) {
    return gt.getChromosome().as(DoubleChromosome.class).toArray();
}
```

6.1.4 Affine transformation

An affine transformation[25], [26] is usually performed by a matrix multiplication with a transformation matrix—in a homogeneous coordinates system[27]. For a

[25] `https://en.wikipedia.org/wiki/Affine_transformation`
[26] `http://mathworld.wolfram.com/AffineTransformation.html`
[27] `https://en.wikipedia.org/wiki/Homogeneous_coordinates`

transformation in $\mathbb{R}^2$, we can define the matrix A[28]:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix}. \tag{6.1}$$

A simple representation can be done by creating a `Genotype` which contains two `DoubleChromosome`s with a length of 3.

```
Genotype.of(
    DoubleChromosome.of(min, max, 3),
    DoubleChromosome.of(min, max, 3)
);
```

The drawback with this kind of encoding is, that we will create a lot of *invalid* (non-affine transformation matrices) during the evolution process, which must be detected and discarded. It is also difficult to find the right parameters for the *min* and *max* values of the `DoubleChromosome`s.

A better approach will be to encode the transformation parameters instead of the transformation matrix. The affine transformation can be expressed by the following parameters:

- s_x – the scale factor in x direction
- s_y – the scale factor in y direction
- t_x – the offset in x direction
- t_y – the offset in y direction
- θ – the rotation angle clockwise around origin
- k_x – shearing parallel to x axis
- k_y – shearing parallel to y axis

This parameters can then be represented by the following `Genotype`.

```
Genotype.of(
    // Scale
    DoubleChromosome.of(sxMin, sxMax),
    DoubleChromosome.of(syMin, syMax),
    // Translation
    DoubleChromosome.of(txMin, txMax),
    DoubleChromosome.of(tyMin, tyMax),
    // Rotation
    DoubleChromosome.of(thMin, thMax),
    // Shear
    DoubleChromosome.of(kxMin, kxMax),
    DoubleChromosome.of(kyMin, kxMax)
)
```

This encoding ensures that no invalid `Genotype` will be created during the evolution process, since the crossover will be only performed on the same kind of

[28] `https://en.wikipedia.org/wiki/Transformation_matrix`

chromosome (same chromosome index). To convert the `Genotype` back to the transformation matrix A, the following equations can be used:

$$\begin{aligned}
a_{11} &= s_x \cos\theta + k_x s_y \sin\theta \\
a_{12} &= s_y k_x \cos\theta - s_x \sin\theta \\
a_{13} &= t_x \\
a_{21} &= k_y s_x \cos\theta + s_y \sin\theta \\
a_{22} &= s_y \cos\theta - s_x k_y \sin\theta \\
a_{23} &= t_y
\end{aligned} \tag{6.2}$$

This corresponds to an transformation order of $T \cdot S_h \cdot S_c \cdot R$:

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & k_x & 0 \\ k_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

In Java code, the conversion from the `Genotype` to the transformation matrix, will look like this:

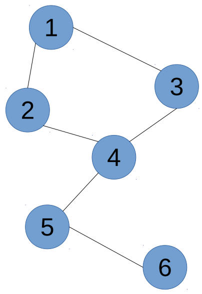
```java
static double[][] toMatrix(final Genotype<DoubleGene> gt) {
    final double sx = gt.get(0, 0).doubleValue();
    final double sy = gt.get(1, 0).doubleValue();
    final double tx = gt.get(2, 0).doubleValue();
    final double ty = gt.get(3, 0).doubleValue();
    final double th = gt.get(4, 0).doubleValue();
    final double kx = gt.get(5, 0).doubleValue();
    final double ky = gt.get(6, 0).doubleValue();

    final double cos_th = cos(th);
    final double sin_th = sin(th);
    final double a11 = cos_th*sx + kx*sy*sin_th;
    final double a12 = cos_th*kx*sy - sx*sin_th;
    final double a21 = cos_th*ky*sx + sy*sin_th;
    final double a22 = cos_th*sy - ky*sx*sin_th;

    return new double[][] {
        {a11, a12, tx},
        {a21, a22, ty},
        {0.0, 0.0, 1.0}
    };
}
```

For the introduced encoding all kind of alterers can be used. Since we have one scalar `DoubleChromosome`, the rotation angle θ, it is recommended also to add an `MeanAlterer` or `GaussianAlterer` to the list of alterers.

6.1.5 Graph

A graph can be represented in many different ways. The most known graph representation is the adjacency matrix. The following encoding examples uses adjacency matrices with different characteristics.

Undirected graph In an undirected graph the edges between the vertices have no direction. If there is a path between nodes i and j, it is assumed that there is also path from j to i.



	1	2	3	4	5	6
1	0	1	1	0	0	0
2		0	0	1	0	0
3			0	1	0	0
4				0	1	0
5					0	1
6						0

Figure 6.1: Undirected graph and adjacency matrix

Figure 6.1 shows an undirected graph and its corresponding matrix representation. Since the edges between the nodes have no direction, the values of the lower diagonal matrix are not taken into account. An application which optimizes an undirected graph has to ignore this part of the matrix.[29]

```java
final int n = 6;
final Genotype<BitGene> gt = Genotype.of(BitChromosome.of(n), n);
```

The code snippet above shows how to create an adjacency matrix for a graph with $n = 6$ nodes. It creates a genotype which consists of n `BitChromosomes` of length n each. Whether the node i is connected to node j can be easily checked by calling `gt.get(i-1, j-1).booleanValue()`. For extracting the whole matrix as `int[]` array, the following code can be used.

```java
final int[][] array = gt.toSeq().stream()
    .map(c -> c.toSeq().stream()
        .mapToInt(BitGene::ordinal)
        .toArray())
    .toArray(int[][]::new);
```

Directed graph A directed graph (digraph) is a graph where the path between the nodes have a direction associated with them. The encoding of a directed graph looks exactly like the encoding of an undirected graph. This time the whole matrix is used and the second diagonal matrix is no longer ignored.

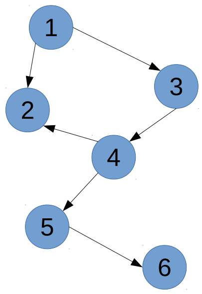
Figure 6.2 on the next page shows the adjacency matrix of a digraph. This time the whole matrix is used for representing the graph.

Weighted directed graph A weighted graph associates a weight (label) with every path in the graph. Weights are usually real numbers. They may be restricted to rational numbers or integers.

The following code snippet shows how the `Genotype` of the matrix is created.

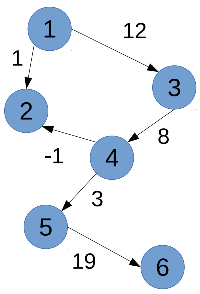
```java
final int n = 6;
final double min = -1;
final double max = 20;
```

[29]This property violates the *minimal* encoding requirement we mentioned at the beginning of section 6.1 on page 43. For simplicity reason this will be ignored for the undirected graph encoding.



	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	0	0	0	0
3	0	0	0	1	0	0
4	0	1	0	0	1	0
5	0	0	0	0	0	1
6	0	0	0	0	0	0

Figure 6.2: Directed graph and adjacency matrix



	1	2	3	4	5	6
1	0	1	12	0	0	0
2	0	0	0	0	0	0
3	0	0	0	8	0	0
4	0	-1	0	0	3	0
5	0	0	0	0	0	19
6	0	0	0	0	0	0

Figure 6.3: Weighted graph and adjacency matrix

```java
final Genotype<DoubleGene> gt = Genotype
    .of(DoubleChromosome.of(min, max, n), n);
```

For accessing the single matrix elements, you can simply call `Genotype.get(i, j).doubleValue()`. If the interaction with another library requires an `double[][]` array, the following code can be used.

```java
final double[][] array = gt.stream()
    .map(dc -> dc.as(DoubleChromosome.class).toArray())
    .toArray(double[][]::new);
```

6.2 Codec

The `Codec` interface—located in the `org.jenetics.engine` package—narrows the gap between the fitness `Function`, which should be maximized/minimized, and the `Genotype` representation, which can be understand by the evolution `Engine`. With the `Codec` interface it is possible to implement the encodings of section 6.1 on page 43 in a more formalized way.

Normally, the `Engine` expects a fitness function which takes a `Genotype` as input. This `Genotype` has then to be *transformed* into an object of the problem domain. The usage `Codec` interface allows a tighter coupling of the `Genotype` definition and the transformation code.[30]

[30] Section 6.1 on page 43 describes some possible encodings for common optimization prob-

```java
public interface Codec<T, G extends Gene<?, G>> {
    public Factory<Genotype<G>> encoding();
    public Function<Genotype<G>, T> decoder();
    public default T decode(final Genotype<G> gt) {...}
}
```

Listing 21: `Codec` interface

Listing 21 shows the `Codec` interface. The `encoding()` method returns the `Genotype` factory, which is used by the `Engine` for creating new `Genotypes`. The decoder `Function`, which is returned by the `decoder()` method, transforms the `Genotype` to the argument type of the fitness `Function`. Without the `Codec` interface, the implementation of the fitness Function is *polluted* with code, which transforms the `Genotype` into the argument type of the actual fitness `Function`.

```java
static double eval(final Genotype<DoubleGene> gt) {
    final double x = gt.getGene().doubleValue();
    // Do some calculation with 'x'.
    return ...
}
```

The `Codec` for the example above is quite simple and is shown below. It is not necessary to implement the `Codec` interface, instead you can use the `Codec.of` factory method for creating new `Codec` instances.

```java
final DoubleRange domain = DoubleRange.of(0, 2*PI);
final Codec<Double, DoubleGene> codec = Codec.of(
    Genotype.of(DoubleChromosome.of(domain)),
    gt -> gt.getChromosome().getGene().getAllele()
);
```

When using an `Codec` instance, the fitness `Function` solely contains code from your actual problem domain—no dependencies to classes of the **Jenetics** library.

```java
static double eval(final double x) {
    // Do some calculation with 'x'.
    return ...
}
```

Jenetics comes with a set of standard encodings, which are created via static factory methods of the `org.jenetics.engine.codecs` class. The following subsections shows some of the implementation of this methods.

6.2.1 Scalar codec

Listing 22 shows the implementation of the `codecs.ofScalar` factory method—for `Integer` scalars.

```java
static Codec<Integer, IntegerGene> ofScalar(IntRange domain) {
    return Codec.of(
        Genotype.of(IntegerChromosome.of(domain)),
        gt -> gt.getChromosome().getGene().getAllele()
    );
}
```

Listing 22: `Codec` factory method: `ofScalar`

The usage of the `Codec`, created by this factory method, simplifies the implementation of the fitness `Function` and the creation of the evolution `Engine`. For

lems.

scalar types, the saving, in complexity and lines of code, is not that big, but using the factory method is still quite handy.

The following listing demonstrates the interaction between `Codec`, fitness `Function` and evolution `Engine`.

```java
class Main {
    // Fitness function directly takes an 'int' value.
    static double fitness(int arg) {
        return ...;
    }
    public static void main(String[] args) {
        final Engine<IntegerGene, Double> engine = Engine
            .builder(Main::fitness, ofScalar(IntRange.of(0, 100)))
            .build();
        ...
    }
}
```

6.2.2 Vector codec

In the listing 23, the `ofVector` factory method returns a `Codec` for an `int[]` array. The `domain` parameter defines the allowed range of the `int` values and the `length` defines the length of the encoded `int` array.

```java
static Codec<int[], IntegerGene> ofVector(
    IntRange domain,
    int length
) {
    return Codec.of(
        Genotype.of(IntegerChromosome.of(domain, length)),
        gt -> gt.getChromosome()
                .as(IntegerChromosome.class)
                .toArray()
    );
}
```

Listing 23: `Codec` factory method: `ofVector`

The usage example of the *vector* `Codec` is almost the same as for the *scalar* `Codec`. As additional parameter, we need to define the length of the desired array and we define our fitness function with an `int[]` array.

```java
class Main {
    // Fitness function directly takes an 'int[]' array.
    static double fitness(int[] args) {
        return ...;
    }
    public static void main(String[] args) {
        final Engine<IntegerGene, Double> engine = Engine
            .builder(
                Main::fitness,
                ofVector(IntRange.of(0, 100), 10))
            .build();
        ...
    }
}
```

6.2.3 Subset codec

There are currently two kinds of subset codecs you can choose from: finding subsets with *variable* size and with *fixed* size.

Variable-sized subsets A Codec for *variable-sized* subsets can be easily implemented with the use of a BitChromosome, as shown in listing 24.

```
static <T> Codec<ISeq<T>, BitGene> ofSubSet(ISeq<T> basicSet) {
    return Codec.of(
        Genotype.of(BitChromosome.of(basicSet.length())),
        gt -> ((BitChromosome)gt.getChromosome()).ones()
            .mapToObj(basicSet::get)
            .collect(ISeq.toISeq())
    );
}
```

Listing 24: `Codec` factory method: `ofSubSet`

The following usage example of *subset* `Codec` shows a simplified version of the Knapsack problem (see section 9.4 on page 80). We try to find a subset, from the given basic `SET`, where the sum of the values is as big as possible, but smaller or equal than 20.

```
class Main {
    // The basic set from where to choose an 'optimal' subset.
    final static ISeq<Integer> SET =
        ISeq.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    // Fitness function directly takes an 'int' value.
    static int fitness(ISeq<Integer> subset) {
        assert(subset.size() <= SET.size());
        final int size = subset.stream().collect(
            Collectors.summingInt(Integer::intValue));
        return size <= 20 ? size : 0;
    }
    public static void main(String[] args) {
        final Engine<BitGene, Double> engine = Engine
            .builder(Main::fitness, ofSubSet(SET))
            .build();
        ...
    }
}
```

Fixed-size subsets[31] The second kind of subset codec allows you to find the *best* subset of a given, fixed size. A classical usage for this encoding is the Subset sum problem[32]:
Given a set (or multi-set) of integers, is there a non-empty subset whose sum is zero? For example, given the set $\{-7, -3, -2, 5, 8\}$*, the answer is yes because the subset* $\{-3$, -2, $5\}$ *sums to zero. The problem is NP-complete*[33].

```
public class SubsetSum
    implements Problem<ISeq<Integer>, EnumGene<Integer>, Integer>
{
    private final ISeq<Integer> _basicSet;
    private final int _size;

    public SubsetSum(ISeq<Integer> basicSet, int size) {
        _basicSet = basicSet;
```

[31] The algorithm for choosing subsets based on a FORTRAN77 version, originally implemented by Albert Nijenhuis, Herbert Wilf. The actual Java implementation is based on the C++ version by John Burkardt.[12], [22]
[32] `https://en.wikipedia.org/wiki/Subset_sum_problem`
[33] *https://en.wikipedia.org/wiki/NP-completeness*

```
        _size = size;
    }

    @Override
    public Function<ISeq<Integer>, Integer> fitness() {
        return subset -> abs(
            subset.stream().mapToInt(Integer::intValue).sum());
    }

    @Override
    public Codec<ISeq<Integer>, EnumGene<Integer>> codec() {
        return codecs.ofSubSet(_basicSet, _size);
    }
}
```

6.2.4 Permutation codec

This kind of codec can be used for problems where optimal solution depends on the order of the input elements. A classical example for such problems is the Knapsack problem (chapter 9.5 on page 82).

```
static <T> Codec<T[], EnumGene<T>> ofPermutation(T... alleles) {
    return Codec.of(
        Genotype.of(PermutationChromosome.of(alleles)),
        gt -> gt.getChromosome().stream()
            .map(EnumGene::getAllele)
            .toArray(length -> (T[])Array.newInstance(
                alleles[0].getClass(), length))
    );
}
```

Listing 25: `Codec` factory method: `ofPermutation`

Listing 25 shows the implementation of a permutation codec, where the order of the given alleles influences the value of the fitness function. An alternate formulation of the traveling salesman problem is shown in the following listing. It uses the permutation codec in listing 25 and uses `java.awt.geom` `Point`s for representing the city locations.

```
public class TSM {
    // The locations to visit.
    static final ISeq<Point> POINTS = ISeq.of(...);

    // The permutation codec.
    static final Codec<ISeq<Point>, EnumGene<Point>>
    CODEC = codecs.ofPermutation(POINTS);

    // The fitness function (in the problem domain).
    static double dist(final ISeq<Point> p) {
        return IntStream.range(0, p.length)
            .mapToDouble(i -> p.get(i)
                .distance(p.get(i + i%p.length())))
            .sum();
    }

    // The evolution engine.
    static final Engine<EnumGene<Point>, Double> ENGINE = Engine
        .builder(TSM::dist, CODEC)
        .optimize(Optimize.MINIMUM)
        .build();

```

```java
    // Find the solution.
    public static void main(final String[] args) {
        final ISeq<Point> result = CODEC.decode(
            ENGINE.stream()
                .limit(10)
                .collect(EvolutionResult.toBestGenotype())
        );

        System.out.println(result);
    }
}
```

6.2.5 Composite codec

The *composite* `Codec` factory method allows to combine two or more `Codecs` into one. Listing 26 shows the method signature of the factory method, which is implemented directly in the `Codec` interface.

```java
static <G extends Gene<?, G>, A, B, T> Codec<T, G> of(
    final Codec<A, G> codec1,
    final Codec<B, G> codec2,
    final BiFunction<A, B, T> decoder
){...}
```

Listing 26: Composite `Codec` factory method

As you can see from the method definition, the combining `Codecs` and the combined `Codec` have the same `Gene` type.

Only `Codecs` which the same `Gene` type can be composed by the combining factory methods of the `Codec` class.

The following listing shows a full example which uses a combined `Codec`. It uses the subset `Codec`, introduced in section 6.2.3 on page 51, and combines it into a `Tuple` of subsets.

```java
class Main {
    static final ISeq<Integer> SET =
        ISeq.of(1, 2, 3, 4, 5, 6, 7, 8, 9);

    // Result type of the combined 'Codec'.
    static final class Tuple<A, B> {
        final A first;
        final B second;
        Tuple(final A first, final B second) {
            this.first = first;
            this.second = second;
        }
    }

    static int fitness(Tuple<ISeq<Integer>, ISeq<Integer>> args) {
        return args.first.stream()
                .mapToInt(Integer::intValue).sum() -
            args.second.stream()
                .mapToInt(Integer::intValue).sum();
    }

    public static void main(String[] args) {
```

```java
        // Combined 'Codec'.
        final Codec<Tuple<ISeq<Integer>, ISeq<Integer>>, BitGene>
            codec = Codec.of(
                codecs.ofSubSet(SET),
                codecs.ofSubSet(SET),
                Tuple::new
            );

        final Engine<BitGene, Integer> engine = Engine
            .builder(Main::fitness, codec)
            .build();

        final Phenotype<BitGene, Integer> pt = engine.stream()
            .limit(100)
            .collect(EvolutionResult.toBestPhenotype());

        // Use the codec for converting the result 'Genotype'.
        final Tuple<ISeq<Integer>, ISeq<Integer>> result =
            codec.decoder().apply(pt.getGenotype());
    }
}
```

If you have to combine more than one Codec into one, you have to use the second, more general, *combining* function: `Codec.of(ISeq<Codec<?, G>>,-Function<Object[], T>)`. The example above shows how to use the general combining function. It is just a little bit more verbose and requires explicit casts for the *sub-codec* types.

```java
final Codec<Triple<Long, Long, Long>, LongGene>
    codec = Codec.of(ISeq.of(
        codecs.ofScalar(LongRange.of(0, 100)),
        codecs.ofScalar(LongRange.of(0, 1000)),
        codecs.ofScalar(LongRange.of(0, 10000))),
        values -> {
            final Long first = (Long)values[0];
            final Long second = (Long)values[1];
            final Long third = (Long)values[2];
            return new Triple<>(first, second, third);
        }
    );
```

6.3 Problem

The `Problem` interface is a further abstraction level, which allows to *bind* the problem encoding and the fitness function into one class.

```java
public interface Problem<
    T,
    G extends Gene<?, G>,
    C extends Comparable<? super C>
> {
    public Function<T, C> fitness();
    public Codec<T, G> codec();
}
```

Listing 27: `Problem` interface

Listing 27 shows the `Problem` interface. The generic type `T` represents the *native* argument type of the fitness function and `C` the `Comparable` result of the fitness function. `G` is the `Gene` type, which is used by the evolution `Engine`.

```java
// Definition of the Ones counting problem.
final Problem<ISeq<BitGene>, BitGene, Integer> ONES_COUNTING =
    Problem.of(
        // Fitness Function<ISeq<BitGene>, Integer>
        genes -> (int)genes.stream()
            .filter(BitGene::getBit).count(),
        Codec.of(
            // Genotype Factory<Genotype<BitGene>>
            Genotype.of(BitChromosome.of(20, 0.15)),
            // Genotype conversion
            // Function<Genotype<BitGene>, <BitGene>>
            gt -> gt.getChromosome().toSeq()
        )
    );

// Engine creation for Problem solving.
final Engine<BitGene, Integer> engine = Engine
    .bulder(ONES_COUNTING)
    .populationSize(150)
    .survivorsSelector(newTournamentSelector<>(5))
    .offspringSelector(new RouletteWheelSelector<>())
    .alterers(
        new Mutator<>(0.03),
        new SinglePointCrossover<>(0.125))
    .build();
```

The listing above shows how a new `Engine` is created by using a predefined `Problem` instance. This allows the complete decoupling of problem and `Engine` definition.

6.4 Validation

A given problem should usually encoded in a way, that it is not possible for the evolution `Engine` to create *invalid* individuals (`Genotypes`). Some possible encodings for common data-structures are described in section 6.1 on page 43. The `Engine` creates new individuals in the *altering* step, by rearranging (or creating new) `Genes` within a `Chromosome`. Since a `Genotype` is treated as *valid* if every single `Gene` in every `Chromosome` is *valid*, the validity property of the `Genes` determines the validity of the whole `Genotype`.

The `Engine` tries to create only valid individuals when creating the initial `Population` and when it replaces `Genotypes` which has been *destroyed* by the altering step. Individuals which has exceeded its lifetime are also replaced by new valid ones. To guarantee the termination of the `Genotype` creation, the Engine is parameterized with the maximal number of retries (`individualCreationRetries`)[34].

If the described validation mechanism doesn't fulfill your needs, you can *override* the validation mechanism by creating the `Engine` with an external `Genotype` *validator*.

```java
final Predicate<? super Genotype<DoubleGene>> validator = gt -> {
    // Implement advanced Genotype check.
    boolean valid = ...;
    return valid;
};
final Engine<DoubleGene, Double> engine = Engine.builder(gtf, ff)
    .limit(100)
```

[34] See section 3.3.3 on page 22.

```
    .genotypeValidator(validator)
    .individualCreationRetries(15)
    .build();
```

Having the possibility to replace the default validation check is a nice thing, but it is better to not create invalid individuals in the first place. For achieving this goal, you have two possibilities:

1. Creating an explicit `Genotype` factory and

2. implementing new `Gene`/`Chromosome`/`Alterer` classes.

`Genotype` factory The usual mechanism for defining an encoding is to create a `Genotype` *prototype*[35]. Since the `Genotype` implements the `Factory` interface, an prototype instance can easily passed to the `Engine.builder` method. For a more advanced `Genotype` creation, you *only* have to create an explicit `Genotype` factory.

```
final Factory<Genotype<DoubleGene>> gtf = () -> {
    // Implement your advanced Genotype factory.
    Genotype<DoubleGene> genotype = ...;
    return genotype;
};
final Engine<DoubleGene, Double> engine = Engine.builder(gtf, ff)
    .limit(100)
    .individualCreationRetries(15)
    .build();
```

With this method you can avoid that the `Engine` creates invalid individuals in the first place, but it is still possible that the alterer step will destroy your `Genotype`s.

`Gene`/`Chromosome`/`Alterer` Creating your own `Gene`, `Chromosome` and `Alterer` classes is the most heavy-wighted possibility for solving the *validity* problem. Refer to section 5 on page 37 for a more detailed description on how to implement this classes.

6.5 Termination

Termination is the criterion by which the evolution stream decides whether to continue or truncate the stream. This section gives a deeper insight into the different ways of terminating or truncating the evolution stream, respectively. The `EvolutionStream` of the **Jenetics** library offers an additional method for limiting the evolution. With the `limit(Predicate<EvolutionResult<G,C>>)` method it is possible to use more advanced termination strategies. If the predicate, given to the `limit` function, returns false, the evolution stream is truncated. `EvolutionStream.limit(r -> true)` will create an infinite evolution stream.

All termination strategies described in the following sub-sections are part of the library and can be created by factory methods of the `org.jenetics.engine.limit` class. The termination strategies where tested by solving the

[35] `https://en.wikipedia.org/wiki/Prototype_pattern`

Knapsack problem[36] (see section 9.4 on page 80) with 250 items. This makes it a real problem with a search-space size of $2^{250} \approx 10^{75}$ elements.

The predicate given to the `EvolutionStream.limit` function must return *false* for truncating the evolution stream. If it returns *true*, the evolution is continued.

Population size:	150
Survivors selector:	`TournamentSelector<>(5)`
Offspring selector:	`RouletteWheelSelector<>()`
Alterers:	`Mutator<>(0.03)` and `SinglePointCrossover<>(0.125)`
Fitness scaler:	Identity function

Table 6.1: Knapsack evolution parameters

Table 6.1 shows the evolution parameters used for the termination tests. To make the tests comparable, all test runs uses the same evolution parameters and the very same set of knapsack items. Each termination test was repeated 1000 times, which gives us enough data to draw the given candlestick diagrams.

Some of the implemented termination strategy needs to maintain an internal state. This strategies can't be re-used in different evolution streams. To be on the safe side, it is recommended to always create an `Predicate` instance for each stream. Calling `Stream.limit(limit.by`*`TerminationStrategy`*`)` will always work as expected.

6.5.1 Fixed generation

The simplest way for terminating the evolution process, is to define a maximal number of generations on the `EvolutionStream`. It just uses the existing `limit` method of the Java `Stream` interface.

```java
final long MAX_GENERATIONS = 100;
EvolutionStream<DoubleGene, Double> stream = engine.stream()
    .limit(MAX_GENERATIONS);
```

This kind of termination method should always be applied—usually additional with other evolution terminators—, to guarantee the truncation of the evolution stream and to define an upper limit of the executed generations.

Figure 6.4 on the following page shows the best fitness values of the used Knapsack problem after a given number of generations, whereas the candlestick points represents the *min*, 25^{th}*percentile*, *median*, 75^{th} *percentile* and *max* fitness after 250 repetitions per generation. The solid line shows for the *mean* of the best fitness values. For a small increase of the fitness value, the needed generations grows exponentially. This is especially the case when the fitness is approaching to its *maximal* value.

[36] The actual implementation used for the termination tests can be found in the Github repository: `https://github.com/jenetics/jenetics/blob/master/org.jenetics.tool/src/main/java/org/jenetics/tool/problem/Knapsack.java`

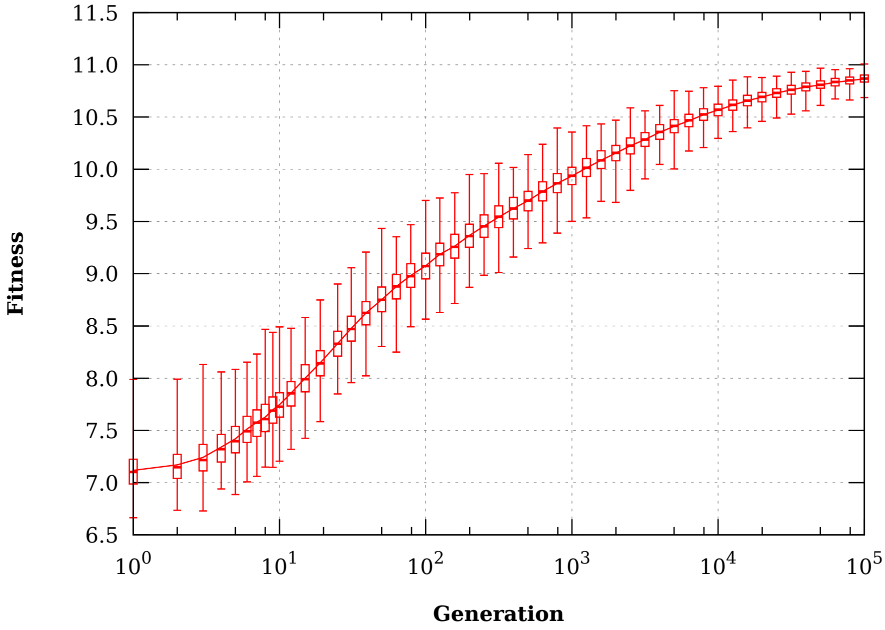


Figure 6.4: Fixed generation termination

6.5.2 Steady fitness

The *steady fitness* strategy truncates the evolution stream if its best fitness hasn't changed after a given number of generations. The predicate maintains an internal state, the number of generations with non increasing fitness, and must be newly created for every evolution stream.

```java
final class SteadyFitnessLimit<C extends Comparable<? super C>>
    implements Predicate<EvolutionResult<?, C>>
{
    private final int _generations;
    private boolean _proceed = true;
    private int _stable = 0;
    private C _fitness;

    public SteadyFitnessLimit(final int generations) {
        _generations = generations;
    }

    @Override
    public boolean test(final EvolutionResult<?, C> er) {
        if (!_proceed) return false;
        if (_fitness == null) {
            _fitness = er.getBestFitness();
            _stable = 1;
        } else {
            final Optimize opt = result.getOptimize();
            if (opt.compare(_fitness, er.getBestFitness()) >= 0) {
                _proceed = ++_stable <= _generations;
            } else {
                _fitness = er.getBestFitness();
                _stable = 1;
```

```
            }
        }
        return _proceed;
    }
}
```

Listing 28: Steady fitness

Listing 28 on the previous page shows the implementation of the `limit.bySteadyFitness(int)` in the `org.jenetics.engine` package. It should give you an impression of how to implement own termination strategies, which possible holds and internal state.

```
Engine<DobuleGene, Double> engine = ...
EvolutionStream<DoubleGene, Double> stream = engine.stream()
    .limit(limit.bySteadyFitness(15));
```

The steady fitness terminator can be created by the `bySteadyFitness` factory method of the `org.jenetics.engine.limit` class. In the example above, the evolution stream is terminated after 15 stable generations.

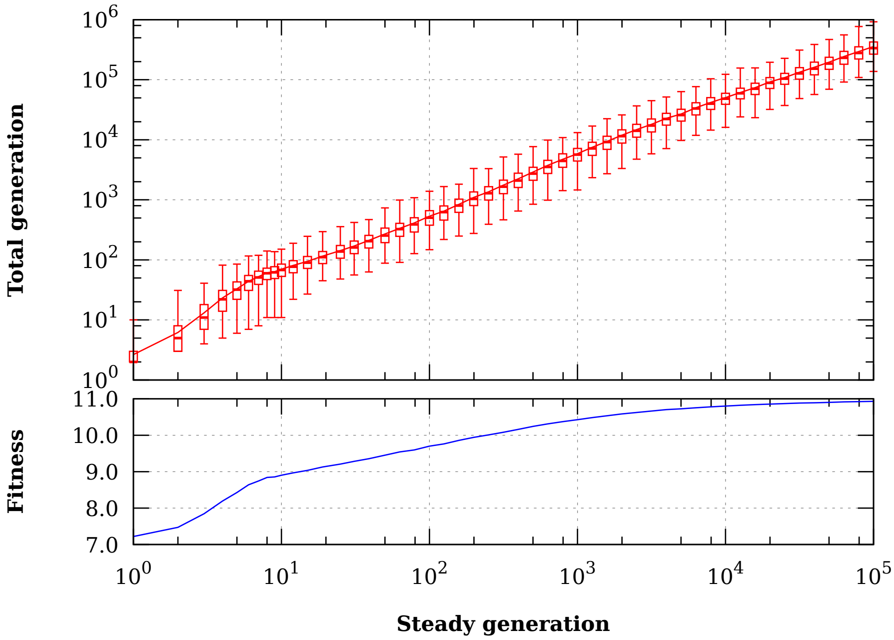


Figure 6.5: Steady fitness termination

Figure 6.5 shows the actual total executed generation depending on the desired number of steady fitness generations. The variation of the total generation is quite big, as shown by the candle-sticks. Though the variation can be quite big—the termination test has been repeated 250 times for each data point—, the tests showed that the *steady fitness* termination strategy always terminated, at least for the given test setup. The lower diagram give an overview of the fitness progression. Only the mean values of the maximal fitness is shown.

6.5.3 Evolution time

This termination strategy stops the evolution when the elapsed evolution time exceeds an user-specified maximal value. The evolution stream is only truncated at the end of an generation and will not interrupt the current evolution step. An maximal evolution time of zero ms will at least evaluate one generation. In an time-critical environment, where a solution must be found within a maximal time period, this terminator let you define the desired guarantees.

```
Engine<DobuleGene, Double> engine = ...
EvolutionStream<DoubleGene, Double> stream = engine.stream()
    .limit(limit.byExecutionTime(Duration.ofMillis(500)));
```

In the code example above, the `byExecutionTime(Duration)` method is used for creating the termination object. Another method, `byExecutionTime(Duration, Clock)`, lets you define the `java.time.Clock`, which is used for measure the execution time. **Jenetics** uses the nano precision clock `org.jenetics.util.NanoClock` for measuring the time. To have the possibility to define a different Clock implementation is especially useful for testing purposes.

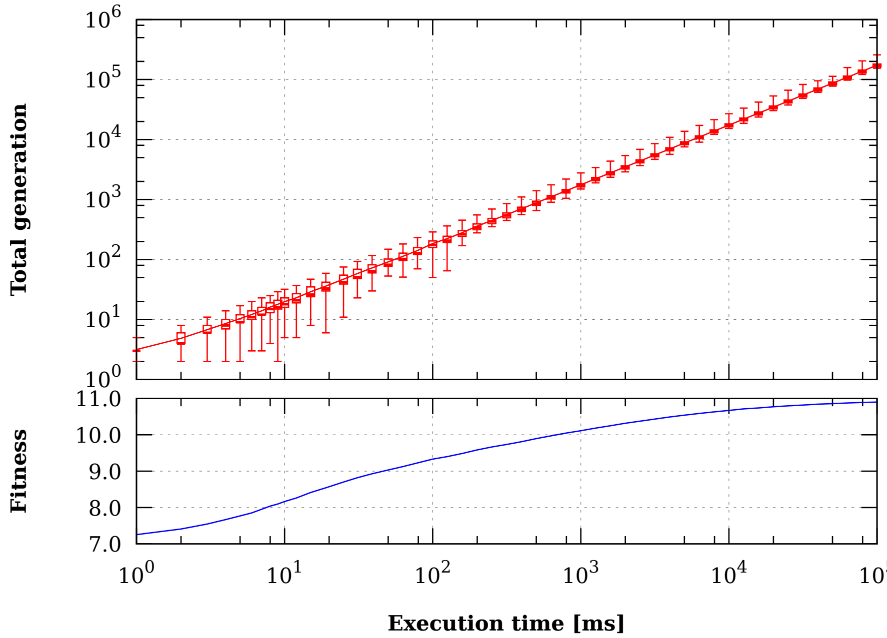


Figure 6.6: Execution time termination

Figure 6.6 shows the evaluated generations depending on the execution time. Except for very small execution times, the evaluated generations per time unit stays quite stable.[37] That means that a doubling of the execution time will double the number of evolved generations.

[37]While running the tests, all other CPU intensive process has been stopped. The measuring started after a warm-up phase.

6.5.4 Fitness threshold

A termination method that stops the evolution when the best fitness in the current population becomes less than the user-specified fitness threshold and the objective is set to minimize the fitness. This termination method also stops the evolution when the best fitness in the current population becomes greater than the user-specified fitness threshold when the objective is to maximize the fitness.

```
Engine<DobuleGene, Double> engine = ...
EvolutionStream<DoubleGene, Double> stream = engine.stream()
    .limit(limit.byFitnessThreshold(10.5)
    .limit(5000);
```

When limiting the evolution stream by a fitness threshold, you have to have a knowledge about the expected maximal fitness. If there is no such knowledge, it is advisable to add an additional fixed sized generation limit as safety net.

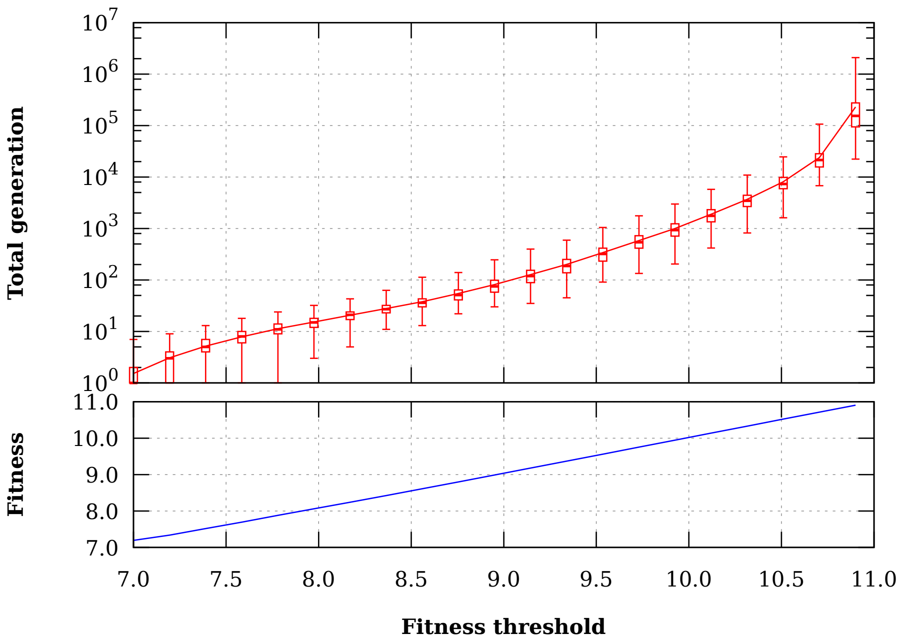


Figure 6.7: Fitness threshold termination

Figure 6.7 shows executed generations depending on the minimal fitness value. The total generations grows exponentially with the desired fitness value. This means, that this termination strategy will (practically) not terminate, if the value for the fitness threshold is chosen to high. And it will definitely not terminate if the fitness threshold is higher than the *global* maximum of the fitness function. It will be a *perfect* strategy if you can define some *good enough* fitness value, which can be *easily* achieved.

6.5.5 Fitness convergence

In this termination strategy, the evolution stops when the fitness is deemed as converged. Two filters of different lengths are used to smooth the best fitness across the generations. When the best smoothed fitness of the long filter is less than a specified percentage away from the best smoothed fitness from the short filter, the fitness is deemed as converged.

Jenetics offers a generic version fitness-convergence predicate and a version where the smoothed fitness is the moving average of the used filters.

```
public static <N extends Number & Comparable<? super N>>
Predicate<EvolutionResult<?, N>> byFitnessConvergence(
    final int shortFilterSize,
    final int longFilterSize,
    final BiPredicate<DoubleMoments, DoubleMoments> proceed
);
```

Listing 29: General fitness convergence

Listing 29 shows the factory method which creates the *generic* fitness convergence predicate. This method allows to define the evolution termination according to the statistical moments of the short- and long fitness filter.

```
public static <N extends Number & Comparable<? super N>>
Predicate<EvolutionResult<?, N>> byFitnessConvergence(
    final int shortFilterSize,
    final int longFilterSize,
    final double epsilon
);
```

Listing 30: Mean fitness convergence

The second factory method (shown in listing 30) creates a fitness convergence predicate, which uses the moving average[38] for the two filters. The smoothed fitness value is calculated as follows:

$$\sigma_F(N) = \frac{1}{N}\sum_{i=0}^{N-1} F_{[G-i]} \tag{6.3}$$

where N is the length of the filter, $F_{[i]}$ the fitness value at generation i and G the current generation. If the condition

$$\frac{|\sigma_F(N_S) - \sigma_F(N_L)|}{\delta} < \epsilon \tag{6.4}$$

is fulfilled, the evolution stream is truncated. Where δ is defined as follows:

$$\delta = \begin{cases} \max\left(|\sigma_F(N_S)|, |\sigma_F(N_L)|\right) & \text{if} \quad \sigma_F(N_x) \neq 0 \\ 1 & \text{otherwise} \end{cases}. \tag{6.5}$$

```
Engine<DobuleGene, Double> engine = ...
EvolutionStream<DoubleGene, Double> stream = engine.stream()
    .limit(limit.byFitnessConvergence(10, 30, 10E-4);
```

For using the fitness convergence strategy you have to specify three parameter. The length of the short filter, N_S, the length of the long filter, N_L and the relative difference between the smoothed fitness values, ϵ.

[38] `https://en.wikipedia.org/wiki/Moving_average`

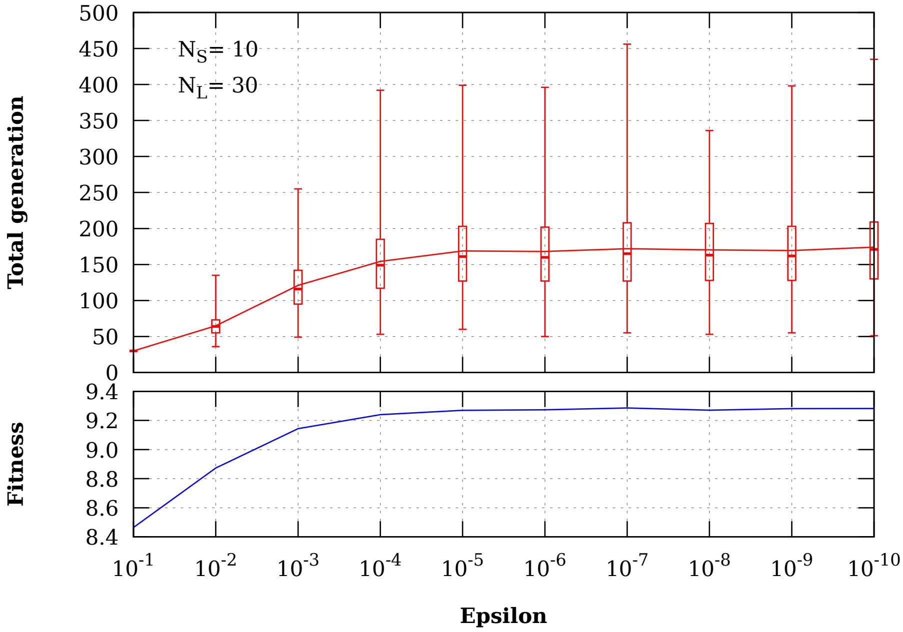


Figure 6.8: Fitness convergence termination: $N_S = 10$, $N_L = 30$

Figure 6.8 shows the termination behavior of the fitness convergence termination strategy. It can be seen that the minimum number of evolved generations is the length of the long filter, N_L.

Figure 6.9 on the following page shows the generations needed for terminating the evolution for higher values of the N_S and N_L parameters.

6.6 Evolution performance

This section contains an empirical *proof*, that *evolutionary* selectors deliver significantly better fitness results than a random search. The `MonteCarloSelector` is used for creating the comparison (random search) fitness values.

Figure 6.10 on page 66 shows the *evolution* performance of the `Selector`[39] used by the examples in section 6.5 on page 57. The lower blue line shows the (mean) fitness values of the *Knapsack* problem when using the `MonteCarloSelector` for selecting the survivors and offspring population. It can be easily seen, that the performance of the *real* evolutionary `Selectors` is much better than a random search.

6.7 Evolution strategies

Evolution Strategies, ES, were developed by Ingo Rechenberg and Hans-Paul Schwefel at the Technical University of Berlin in the mid 1960s.[18] It is a global optimization algorithm in continuous search spaces and is an instance

[39]The termination tests are using a `TournamentSelector`, with tournament-size 5, for selecting the survivors, and a `RouletteWheelSelector` for selecting the offspring.

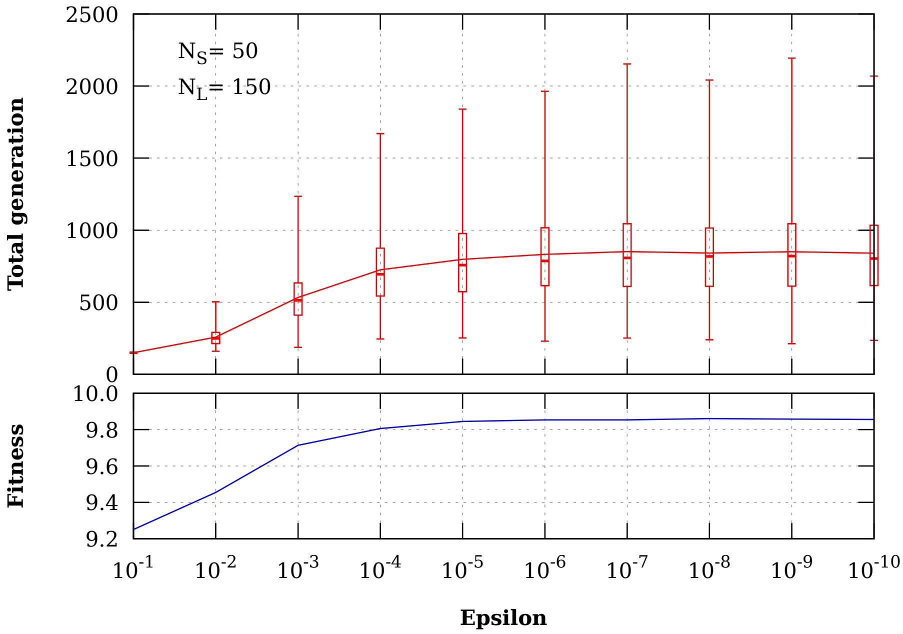


Figure 6.9: Fitness convergence termination: $N_S = 50$, $N_L = 150$

of an *Evolutionary Algorithm* from the field of *Evolutionary Computation*. ES uses truncation selection[40] for selecting the individuals and usually mutation[41] for changing the next generation. This section describes how to configure the evolution `Engine` of the library for the (μ, λ)- and $(\mu + \lambda)$-ES.

6.7.1 (μ, λ) evolution strategy

The (μ, λ) algorithm starts by generating λ individuals randomly. After evaluating the fitness of all the individuals, all but the μ fittest ones are deleted. Each of the μ fittest individuals gets to produce $\frac{\lambda}{\mu}$ children through an ordinary mutation. The newly created children just replaces the discarded parents.[8]

To summarize it: μ is the number of parents which survive, and λ is the number of offspring, created by the μ parents. The value of λ should be a multiple of μ. ES practitioners usually refer to their algorithm by the choice of μ and λ. If we set $\mu = 5$ and $\lambda = 5$, then we have a $(5, 20)$-ES.

```java
final Engine<DoubleGene, Double> engine =
    Engine.builder(fitness, codec)
        .populationSize(lambda)
        .survivorsSize(0)
        .offspringSelector(new TruncationSelector<>(mu))
        .alterers(new Mutator<>(p))
        .build();
```

Listing 31: (μ, λ) `Engine` configuration

[40] See 3.2.1 on page 12.
[41] See 3.2.2 on page 15.

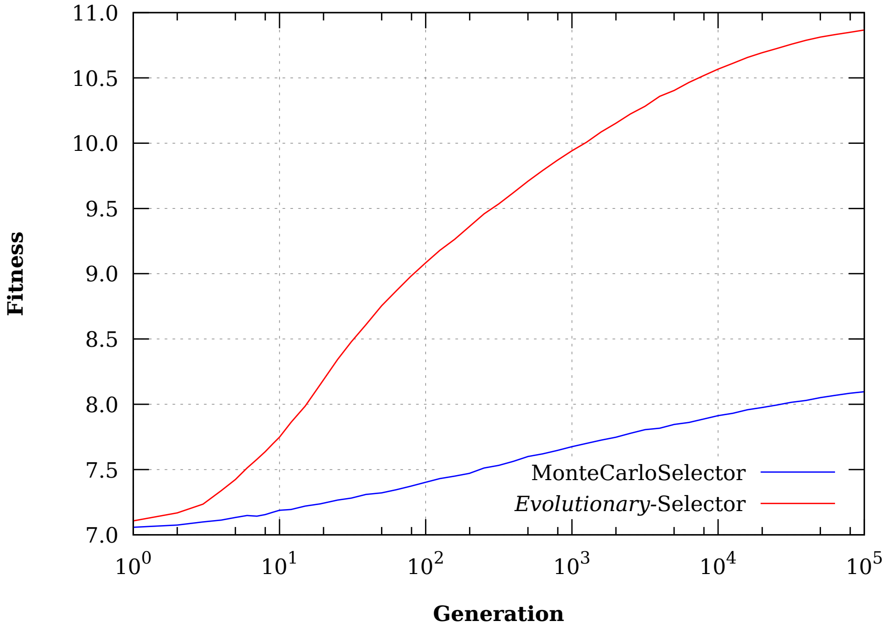


Figure 6.10: Selector-performance (Knapsack)

Listing 31 on the previous page shows how to configure the evolution Engine for (μ, λ)-ES. The population size is set to λ and the survivors size to zero, since the best parents are not part of the final population. Step three is configured by setting the offspring selector to the `TruncationSelector`. Additionally, the `TruncationSelector` is parameterized with μ. This lets the `TruncationSelector` only select the μ best individuals, which corresponds to step two of the ES.

There are mainly three levers for the (μ, λ)-ES where we can adjust exploration versus exploitation:[8]

- **Population size** λ: This parameter controls the sample size for each population. For the extreme case, as λ approaches ∞, the algorithm would perform a simple random search.
- **Survivors size of** μ: This parameter controls how selective the ES is. Relatively low μ values pushes the algorithm towards exploitative search, because only the best individuals are used for reproduction.[42]
- **Mutation probability** p: A high mutation probability pushes the algorithm toward a fairly random search, regardless of the selectivity of μ.

[42] As you can see in listing 31 on the preceding page, the survivors size (reproduction pool size) for the (μ, λ)-ES must be set *indirectly* via the `TruncationSelector` parameter. This is necessary, since for the (μ, λ)-ES, the selected best μ individuals are not part of the population of the next generation.

6.7.2 $(\mu + \lambda)$ evolution strategy

In the $(\mu + \lambda)$-*ES*, the next generation consists of the selected best μ parents and the λ new children. This is also the main difference to (μ, λ), where the μ parents are not part of the next generation. Thus the next and all successive generations are $\mu + \lambda$ in size.[8]

Jenetics works with a constant population size and it is therefore not possible to implement an increasing population size. Besides this restriction, the `Engine` configuration for the $(\mu + \lambda)$-ES is shown in listing 32.

```java
final Engine<DoubleGene, Double> engine =
    Engine.builder(fitness, codec)
        .populationSize(lambda)
        .survivorsSize(mu)
        .selector(new TruncationSelector<>(mu))
        .alterers(new Mutator<>(p))
        .build();
```

Listing 32: $(\mu + \lambda)$ `Engine` configuration

Since the selected μ parents are part of the next generation, the `survivorsSize` property must be set to μ. This also requires to set the survivors selector to the `TruncationSelector`. With the `selector(Selector)` method, both selectors, the selector for the survivors and for the offspring, can be set.

Because the best parents are also part of the next generation, the $(\mu + \lambda)$-ES may be more exploitative than the (μ, λ)-ES. This has the risk, that very fit parents can defeat other individuals over and over again, which leads to a prematurely convergence to a local optimum.

7 Internals

This section contains internal implementation details which doesn't fit in one of the previous sections. They are not essential for using the library, but would give the user a deeper insight in some design decisions, made when implementing the library. It also introduces tools and classes which where developed for testing purpose. This classes resides below the `org.jenetics.internal` package. Though they are **not** part of the official API, they are packed into the delivered jar and can be used accordingly. *Be aware that all classes below the `org.jenetics.internal` package can be changed and removed without announcement.*

7.1 PRNG testing

Jenetics uses the `dieharder`[43] (command line) tool for testing the *randomness* of the used PRNGs. `dieharder` is a random number generator (RNG) testing suite. It is intended to test generators, not files of possibly random numbers. Since `dieharder` needs a huge amount of random data, for testing the quality of a RNG, it is usually advisable to pipe the random numbers to the `dieharder` process:

```
$ cat /dev/urandom | dieharder -g 200 -a
```

[43] From Robert G. Brown: `http://www.phy.duke.edu/~rgb/General/dieharder.php`

The example above demonstrates how to stream a raw binary stream of bits to the `stdin` (raw) interface of `dieharder`. With the `DieHarder` class, which is part of the `org.jenetics.internal.util` package, it is easily possible to test PRNGs extending the `java.util.Random` class. The only requirement is, that the PRNG must be *default*-constructible and part of the classpath.

```
$ java -cp org.jenetics-3.8.0.jar \
      org.jenetics.internal.util.DieHarder \
      <random-engine-name> -a
```

Calling the command above will create an instance of the given random engine and stream the random data (bytes) to the raw interface of `dieharder` process.

```
#=============================================================================#
# Testing: <random-engine-name> (2015-07-11 23:48)                            #
#=============================================================================#
#=============================================================================#
# Linux 3.19.0-22-generic (amd64)                                             #
# java version "1.8.0_45"                                                     #
# Java(TM) SE Runtime Environment (build 1.8.0_45-b14)                        #
# Java HotSpot(TM) 64-Bit Server VM (build 25.45-b02)                         #
#=============================================================================#
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
stdin_input_raw|  1.36e+07  |1583496496|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.63372078|  PASSED
      diehard_operm5|   0|   1000000|     100|0.42965082|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.95159380|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.70376799|  PASSED
...
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.76563780|  PASSED
#=============================================================================#
# Summary: PASSED=112, WEAK=2, FAILED=0                                       #
#          235,031.492 MB of random data created with 41.394 MB/sec           #
#=============================================================================#
#=============================================================================#
# Runtime: 1:34:37                                                            #
#=============================================================================#
```

In the listing above, a part of the created `dieharder` report is shown. For testing the `LCG64ShiftRandom` class, which is part of the `org.jenetics.util` package, the following command can be called:

```
$ java -cp org.jenetics-3.8.0.jar \
      org.jenetics.internal.util.DieHarder \
      org.jenetics.util.LCG64ShiftRandom -a
```

Table 7.1 on the following page shows the summary of the `dieharder` tests. The full report is part of the source file of the `LCG64ShiftRandom` class.[44]

7.2 Random seeding

The PRNGs[45], used by the **Jenetics** library, needs to be initialized with a proper seed value before they can be used. The usual way for doing this, is to take the current time stamp.

[44] `https://github.com/jenetics/jenetics/blob/master/org.jenetics/src/main/java/org/jenetics/util/LCG64ShiftRandom.java`

[45] See section 4.2 on page 30.

Passed tests	Weak tests	Failed tests
110	4	0

Table 7.1: `LCG64ShiftRandom` quality

```java
public static long seed() {
    return System.nanoTime();
}
```

Before applying this method throughout the whole library, I decided to perform some statistical tests. For this purpose I treated the `seed()` method itself as PRNG and analyzed the created long values with the `DieHarder class`. The `seed()` method has been wrapped into the `org.jenetics.internal.util.NanoTimeRandom` class. Assuming that the `dieharder` tool is in the search path, calling

```
$ java -cp org.jenetics-3.8.0.jar \
      org.jenetics.internal.util.DieHarder \
      org.jenetics.internal.util.NanoTimeRandom -a
```

will perform the statistical tests for the nano time *random engine*. The statistical quality is rather bad: every single test failed. Table 7.2 shows the summary of the `dieharder` report.[46]

Passed tests	Weak tests	Failed tests
0	0	114

Table 7.2: Nano time seeding quality

An alternative source of entropy, for generating seed values, would be the `/dev/random` or `/dev/urandom` file. But this approach is not portable, which was a prerequisite for the **Jenetics** library.

The next attempt tries to fetch the seeds from the JVM, via the `Object.hashCode()` method. Since the hash code of an Object is available for every operating system and most likely »randomly« distributed.

```java
public static long seed() {
    return ((long)new Object().hashCode() << 32) |
        new Object().hashCode();
}
```

This seed method has been wrapped into the `ObjectHashRandom` class and tested as well with

```
$ java -cp org.jenetics-3.8.0.jar \
      org.jenetics.internal.util.DieHarder \
      org.jenetics.internal.util.ObjectHashRandom -a
```

Table 7.3 shows the summary of the `dieharder` report[47], which looks better than the nano time seeding, but 86 failing tests was still not very satisfying.

[46] The detailed test report can be found in the source of the NanoTimeRandom class. `https://github.com/jenetics/jenetics/blob/master/org.jenetics/src/main/java/org/jenetics/internal/util/NanoTimeRandom.java`

[47] Full report: `https://github.com/jenetics/jenetics/blob/master/org.jenetics/src/main/java/org/jenetics/internal/util/ObjectHashRandom.java`

Passed tests	Weak tests	Failed tests
28	0	86

Table 7.3: Object hash seeding quality

After additional experimentation, a combination of the nano time seed and the object hash seeding seems to be the *right* solution. The rational behind this was, that the PRNG seed shouldn't rely on a single *source* of entropy.

```java
public static long seed() {
    return mix(System.nanoType(), objectHashSeed());
}

private static long mix(final long a, final long b) {
    long c = a^b;
    c ^= c << 17;
    c ^= c >>> 31;
    c ^= c << 8;
    return c;
}

private static long objectHashSeed() {
    return ((long)new Object().hashCode() << 32) |
        new Object().hashCode();
}
```

Listing 33: Random seeding

The code in listing 33 shows how the nano time seed is mixed with the object seed. The `mix` method was inspired by the mixing step of the `lcg64_shift`[48] random engine, which has been reimplemented in the `LCG64ShiftRandom` class. Running the tests with

```
$ java -cp org.jenetics-3.8.0.jar \
      org.jenetics.internal.util.DieHarder \
      org.jenetics.internal.util.SeedRandom -a
```

leads to the statistics summary[49], which is shown in table 7.4.

Passed tests	Weak tests	Failed tests
112	2	0

Table 7.4: Combined random seeding quality

The statistical performance of this seeding is better, according to the `dieharder` test suite, than some of the real random engines, including the default Java `Random` engine. Using the proposed `seed()` method is in any case preferable to the simple `System.nanoTime()` call.

Open questions

[48]This class is part of the TRNG library: `https://github.com/rabauke/trng4/blob/master/src/lcg64_shift.hpp`

[49]Full report: `https://github.com/jenetics/jenetics/blob/master/org.jenetics/src/main/java/org/jenetics/internal/util/SeedRandom.java`

- How does this method perform on operating systems other than Linux?
- How does this method perform on other JVM implementations?

8 Incubation

This section describes the classes in the *not yet* released modules. Incubating features and experimental genetic operators will be implemented in this modules. If you find this classes useful, you must build the module yourself, since they are not yet available in the global maven repository. With

```
$ ./gradlew <module>:jar
```

it is possible to create a module JAR which can be added to your project classpath. Be aware that interface and/or implementation of incubating modules can be changed without noticing.

Currently incubation modules:

`org.jenetics.tool` Contains utility classes for measuring the evolution performance.[50] This classes where used for creating the diagrams in this manual.

`org.jenetix` Contains non-standard selector and mutator classes.[51] The classes needed for the *Weasel program* (see section 8.1) are part of this module.

8.1 Weasel program

The Weasel program[52] is thought experiment from Richard Dawkins, in which he tries to illustrate the function of genetic *mutation* and *selection*.[53] For this reason he chooses the well known example of typewriting monkeys.

> I don't know who it was first pointed out that, given enough time, a monkey bashing away at random on a typewriter could produce all the works of Shakespeare. The operative phrase is, of course, given enough time. Let us limit the task facing our monkey somewhat. Suppose that he has to produce, not the complete works of Shakespeare but just the short sentence »Methinks it is like a weasel«, and we shall make it relatively easy by giving him a typewriter with a restricted keyboard, one with just the 26 (uppercase) letters, and a space bar. How long will he take to write this one little sentence?[7]

The search space of the 28 character long target string is $27^{28} \approx 10^{40}$. If the monkey writes $1,000,000$ different *sentences* per second, it would take about 10^{26} years (in average) writing the correct one. Although Dawkins did not provide the source code for his program, a »Weasel« style algorithm could run as follows:

1. Start with a random string of 28 characters.

[50] `http://jenetics.io/javadoc/org.jenetics.tool/3.8/index.html`
[51] `http://jenetics.io/javadoc/org.jenetix/3.8/index.html`
[52] `https://en.wikipedia.org/wiki/Weasel_program`
[53] The classes are located in the `org.jenetix` module.

2. Make n copies of the string (reproduce).
3. Mutate the characters with an mutation probability of 5%.
4. Compare each new string with the target string »METHINKS IT IS LIKE A WEASEL«, and give each a score (the number of letters in the string that are correct and in the correct position).
5. If any of the new strings has a perfect score (28), halt. Otherwise, take the highest scoring string, and go to step 2.

Richard Dawkins was also very careful to point out the limitations of this simulation:

> Although the monkey/Shakespeare model is useful for explaining the distinction between single-step selection and cumulative selection, it is misleading in important ways. One of these is that, in each generation of selective »breeding«, the mutant »progeny« phrases were judged according to the criterion of resemblance to a distant ideal target, the phrase METHINKS IT IS LIKE A WEASEL. Life isn't like that. Evolution has no long-term goal. There is no long-distance target, no final perfection to serve as a criterion for selection, although human vanity cherishes the absurd notion that our species is the final goal of evolution. In real life, the criterion for selection is always short-term, either simple survival or, more generally, reproductive success.[7]

If you want to write a Weasel program with the **Jenetics** library, you need to use the special `WeaselSelector` and `WeaselMutator`.

```java
public class WeaselProgram {
    private static final String TARGET =
        "METHINKS IT IS LIKE A WEASEL";

    private static int score(final Genotype<CharacterGene> gt) {
        final CharSequence source =
            (CharSequence)gt.getChromosome();
        return IntStream.range(0, TARGET.length())
            .map(i -> source.charAt(i) == TARGET.charAt(i) ? 1 : 0)
            .sum();
    }

    public static void main(final String[] args) {
        final CharSeq chars = CharSeq.of("A-Z ");
        final Factory<Genotype<CharacterGene>> gtf = Genotype.of(
            new CharacterChromosome(chars, TARGET.length())
        );
        final Engine<CharacterGene, Integer> engine = Engine
            .builder(WeaselProgram::score, gtf)
            .populationSize(150)
            .selector(new WeaselSelector<>())
            .offspringFraction(1)
            .alterers(new WeaselMutator<>(0.05))
            .build();
        final Phenotype<CharacterGene, Integer> result = engine
            .stream()
            .limit(byFitnessThreshold(TARGET.length() - 1))
            .peek(r -> System.out.println(
```

```
                    r.getTotalGenerations() + ": " +
                    r.getBestPhenotype()))
                .collect(toBestPhenotype());
            System.out.println(result);
        }
}
```

Listing 34: `Weasel program`

Listing 34 on the previous page shows how-to implement the `WeaselProgram` with **Jenetics**. Step (1) and (2) of the algorithm is done implicitly when the initial population is created. The third step is done by the `WeaselMutator`, with mutation probability of 0.05. Step (4) is done by the WeaselSelector together with the configured offspring-fraction of one. The evolution stream is limited by the `limit.byFitnessThreshold`, which is set to $score_{max} - 1$. In the current example this value is set to `TARGET.length() - 1` = 27.

```
1:  [UBNHLJUS RCOXR LFIYLAWRDCCNY] --> 6
2:  [UBNHLJUS RCOXR LFIYLAWWDCCNY] --> 7
3:  [UBQHLJUS RCOXR LFIYLAWWECCNY] --> 8
5:  [UBQHLJUS RCOXR LFICLAWWECCNL] --> 9
6:  [W QHLJUS RCOXR LFICLA WEGCNL] --> 10
7:  [W QHLJKS RCOXR LFIHLA WEGCNL] --> 11
8:  [W QHLJKS RCOXR LFIHLA WEGSNL] --> 12
9:  [W QHLJKS RCOXR LFIS A WEGSNL] --> 13
10: [M QHLJKS RCOXR LFIS A WEGSNL] --> 14
11: [MEQHLJKS RCOXR LFIS A WEGSNL] --> 15
12: [MEQHIJKS ICOXR LFIN A WEGSNL] --> 17
14: [MEQHINKS ICOXR LFIN A WEGSNL] --> 18
16: [METHINKS ICOXR LFIN A WEGSNL] --> 19
18: [METHINKS IMOXR LFKN A WEGSNL] --> 20
19: [METHINKS IMOXR LIKN A WEGSNL] --> 21
20: [METHINKS IMOIR LIKN A WEGSNL] --> 22
23: [METHINKS IMOIR LIKN A WEGSEL] --> 23
26: [METHINKS IMOIS LIKN A WEGSEL] --> 24
27: [METHINKS IM IS LIKN A WEHSEL] --> 25
32: [METHINKS IT IS LIKN A WEHSEL] --> 26
42: [METHINKS IT IS LIKN A WEASEL] --> 27
46: [METHINKS IT IS LIKE A WEASEL] --> 28
```

The (shortened) output of the Weasel program (listing 34 on the preceding page) shows, that the optimal solution is reached in generation 46.

Appendix

9 Examples

This section contains some coding examples which should give you a feeling of how to use the **Jenetics** library. The given examples are complete, in the sense that they will compile and run and produce the given example output.

Running the examples delivered with the **Jenetics** library can be started with the `run-examples.sh` script.

```
$ ./run-examples.sh
```

Since the script uses JARs located in the build directory you have to build it with the `jar` *Gradle* target first; see section 10 on page 86.

9.1 Ones counting

Ones counting is one of the simplest model-problem. It uses a binary chromosome and forms a classic genetic algorithm[54]. The fitness of a `Genotype` is proportional to the number of ones.

```java
import static org.jenetics.engine.EvolutionResult.toBestPhenotype;
import static org.jenetics.engine.limit.bySteadyFitness;

import org.jenetics.BitChromosome;
import org.jenetics.BitGene;
import org.jenetics.Genotype;
import org.jenetics.Mutator;
import org.jenetics.Phenotype;
import org.jenetics.RouletteWheelSelector;
import org.jenetics.SinglePointCrossover;
import org.jenetics.engine.Engine;
import org.jenetics.engine.EvolutionStatistics;

public class OnesCounting {

    // This method calculates the fitness for a given genotype.
    private static Integer count(final Genotype<BitGene> gt) {
        return gt.getChromosome()
            .as(BitChromosome.class)
            .bitCount();
    }

    public static void main(String[] args) {
        // Configure and build the evolution engine.
        final Engine<BitGene, Integer> engine = Engine
            .builder(
                OnesCounting::count,
                BitChromosome.of(20, 0.15))
            .populationSize(500)
            .selector(new RouletteWheelSelector<>())
            .alterers(
                new Mutator<>(0.55),
                new SinglePointCrossover<>(0.06))
            .build();
```

[54] In the classic genetic algorithm the problem is a maximization problem and the fitness function is positive. The domain of the fitness function is a bit-chromosome.

```java

        // Create evolution statistics consumer.
        final EvolutionStatistics<Integer, ?>
            statistics = EvolutionStatistics.ofNumber();

        final Phenotype<BitGene, Integer> best = engine.stream()
            // Truncate the evolution stream after 7 "steady"
            // generations.
            .limit(bySteadyFitness(7))
            // The evolution will stop after maximal 100
            // generations.
            .limit(100)
            // Update the evaluation statistics after
            // each generation
            .peek(statistics)
            // Collect (reduce) the evolution stream to
            // its best phenotype.
            .collect(toBestPhenotype());

        System.out.println(statistics);
        System.out.println(best);
    }
}
```

The genotype in this example consists of one `BitChromosome` with a ones probability of 0.15. The altering of the offspring population is performed by mutation, with mutation probability of 0.55, and then by a single-point crossover, with crossover probability of 0.06. After creating the initial population, with the `ga.setup()` call, 100 generations are evolved. The tournament selector is used for both, the offspring- and the survivor selection—this is the default selector.[55]

```
+---------------------------------------------------------------------------+
|  Time statistics                                                          |
+---------------------------------------------------------------------------+
|             Selection: sum=0.016580144000 s; mean=0.001381678667 s        |
|              Altering: sum=0.096904159000 s; mean=0.008075346583 s        |
|   Fitness calculation: sum=0.022894318000 s; mean=0.001907859833 s        |
|     Overall execution: sum=0.136575323000 s; mean=0.011381276917 s        |
+---------------------------------------------------------------------------+
|  Evolution statistics                                                     |
+---------------------------------------------------------------------------+
|           Generations: 12                                                 |
|               Altered: sum=40,487; mean=3373.916666667                    |
|                Killed: sum=0; mean=0.000000000                            |
|              Invalids: sum=0; mean=0.000000000                            |
+---------------------------------------------------------------------------+
|  Population statistics                                                    |
+---------------------------------------------------------------------------+
|                     Age: max=9; mean=0.808667; var=1.446299               |
|               Fitness:                                                    |
|                      min  = 1.000000000000                                |
|                      max  = 18.000000000000                               |
|                      mean = 10.050833333333                               |
|                      var  = 7.839555898205                                |
|                      std  = 2.799920694985                                |
+---------------------------------------------------------------------------+
[00001101|11110111|11111111] --> 18
```

The given example will print the overall timing statistics onto the console. In the *Evolution statistics* section you can see that it actually takes 15 generations to fulfill the termination criteria—finding no better result after 7 consecutive generations.

[55]For the other default values (population size, maximal age, ...) have a look at the Javadoc: `http://jenetics.io/javadoc/org.jenetics/3.8/index.html`

9.2 Real function

In this example we try to find the minimum value of the function

$$f(x) = \cos\left(\frac{1}{2} + \sin(x)\right) \cdot \cos(x). \tag{9.1}$$

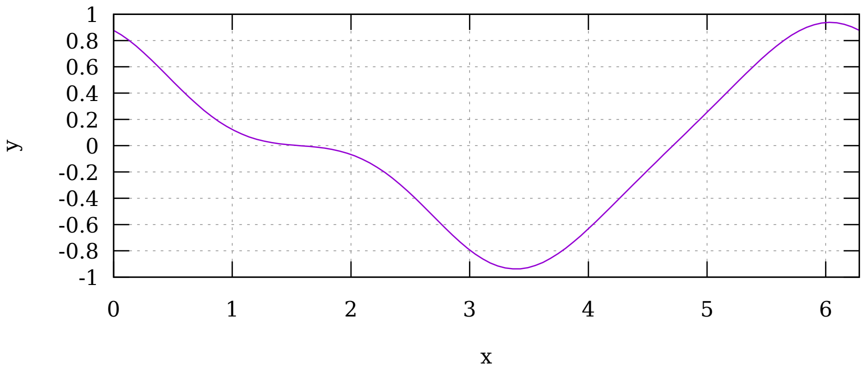


Figure 9.1: Real function

The graph of function 9.1, in the range of $[0, 2\pi]$, is shown in figure 9.1 and the listing beneath shows the GA implementation which will minimize the function.

```java
import static java.lang.Math.PI;
import static java.lang.Math.cos;
import static java.lang.Math.sin;
import static org.jenetics.engine.EvolutionResult.toBestPhenotype;
import static org.jenetics.engine.limit.bySteadyFitness;

import org.jenetics.DoubleGene;
import org.jenetics.MeanAlterer;
import org.jenetics.Mutator;
import org.jenetics.Optimize;
import org.jenetics.Phenotype;
import org.jenetics.engine.Engine;
import org.jenetics.engine.EvolutionStatistics;
import org.jenetics.engine.codecs;
import org.jenetics.util.DoubleRange;

public class RealFunction {

    // The fitness function.
    private static double fitness(final double x) {
        return cos(0.5 + sin(x))*cos(x);
    }

    public static void main(final String[] args) {
        final Engine<DoubleGene, Double> engine = Engine
            // Create a new builder with the given fitness
            // function and chromosome.
            .builder(
                RealFunction::fitness,
                codecs.ofScalar(DoubleRange.of(0.0, 2.0*PI)))
```

```java
                .populationSize(500)
                .optimize(Optimize.MINIMUM)
                .alterers(
                    new Mutator<>(0.03),
                    new MeanAlterer<>(0.6))
                // Build an evolution engine with the
                // defined parameters.
                .build();

        // Create evolution statistics consumer.
        final EvolutionStatistics<Double, ?>
            statistics = EvolutionStatistics.ofNumber();

        final Phenotype<DoubleGene, Double> best = engine.stream()
            // Truncate the evolution stream after 7 "steady"
            // generations.
            .limit(bySteadyFitness(7))
            // The evolution will stop after maximal 100
            // generations.
            .limit(100)
            // Update the evaluation statistics after
            // each generation
            .peek(statistics)
            // Collect (reduce) the evolution stream to
            // its best phenotype.
            .collect(toBestPhenotype());

        System.out.println(statistics);
        System.out.println(best);
    }
}
```

The GA works with 1×1 `DoubleChromosomes` whose values are restricted to the range $[0, 2\pi]$.

```
+---------------------------------------------------------------------------+
|  Time statistics                                                          |
+---------------------------------------------------------------------------+
|             Selection: sum=0.064406456000 s; mean=0.003066974095 s       |
|              Altering: sum=0.070158382000 s; mean=0.003340875333 s       |
|   Fitness calculation: sum=0.050452647000 s; mean=0.002402507000 s       |
|     Overall execution: sum=0.169835154000 s; mean=0.008087388286 s       |
+---------------------------------------------------------------------------+
|  Evolution statistics                                                     |
+---------------------------------------------------------------------------+
|           Generations: 21                                                 |
|               Altered: sum=3,897; mean=185.571428571                      |
|                Killed: sum=0; mean=0.000000000                            |
|              Invalids: sum=0; mean=0.000000000                            |
+---------------------------------------------------------------------------+
|  Population statistics                                                    |
+---------------------------------------------------------------------------+
|                   Age: max=9; mean=1.104381; var=1.962625                 |
|               Fitness:                                                    |
|                      min  = -0.938171897696                               |
|                      max  = 0.936310125279                                |
|                      mean = -0.897856583665                               |
|                      var  = 0.027246274838                                |
|                      std  = 0.165064456617                                |
+---------------------------------------------------------------------------+
[[[3.389125782657314]]] --> -0.9381718976956661
```

The GA will generated an console output like above. The *exact* result of the function–for the given range–will be 3.389, 125, 782, 8907, 939... You can also see, that we reached the final result after 19 generations.

9.3 Rastrigin function

The Rastrigin function[56] is often used to test the optimization performance of genetic algorithm.

$$f(\mathbf{x}) = An + \sum_{i=1}^{n} \left(x_i^2 - A \cos(2\pi x_i) \right). \tag{9.2}$$

As the plot in figure 9.2 shows, the Rastrigin function has many local minima, which makes it difficult for standard, gradient-based methods to find the global minimum. If $A = 10$ and $x_i \in [-5.12, 5.12]$, the function has only one global minimum at $\mathbf{x} = \mathbf{0}$ with $f(\mathbf{x}) = 0$.

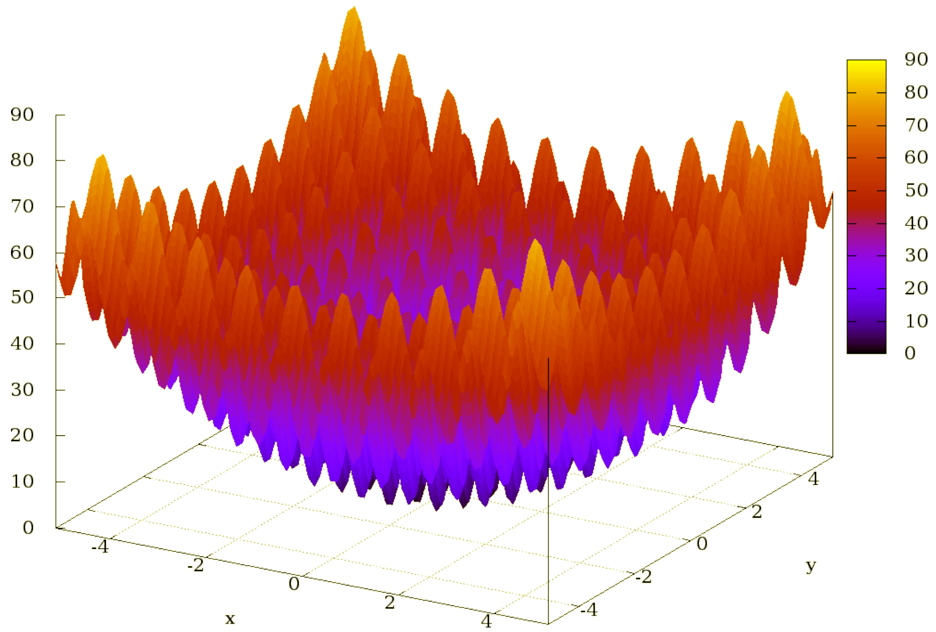


Figure 9.2: Rastrigin function

The following listing shows the `Engine` setup for solving the Rastrigin function, which is very similar to the setup for the real-function in section 9.2 on page 76. Beside the different fitness function, the `Codec` for `double` vectors is used, instead of the `double` scalar `Codec`.

```java
import static java.lang.Math.PI;
import static java.lang.Math.cos;
import static org.jenetics.engine.EvolutionResult.toBestPhenotype;
import static org.jenetics.engine.limit.bySteadyFitness;

import org.jenetics.DoubleGene;
import org.jenetics.MeanAlterer;
import org.jenetics.Mutator;
import org.jenetics.Optimize;
import org.jenetics.Phenotype;
import org.jenetics.engine.Engine;
```

56 `https://en.wikipedia.org/wiki/Rastrigin_function`

```java
import org.jenetics.engine.EvolutionStatistics;
import org.jenetics.engine.codecs;
import org.jenetics.util.DoubleRange;

public class RastriginFunction {
    private static final double A = 10;
    private static final double R = 5.12;
    private static final int N = 2;

    private static double fitness(final double[] x) {
        double value = A*N;
        for (int i = 0; i < N; ++i) {
            value += x[i]*x[i] - A*cos(2.0*PI*x[i]);
        }

        return value;
    }

    public static void main(final String[] args) {
        final Engine<DoubleGene, Double> engine = Engine
            .builder(
                RastriginFunction::fitness,
                // Codec for 'x' vector.
                codecs.ofVector(DoubleRange.of(-R, R), N))
            .populationSize(500)
            .optimize(Optimize.MINIMUM)
            .alterers(
                new Mutator<>(0.03),
                new MeanAlterer<>(0.6))
            .build();

        final EvolutionStatistics<Double, ?>
            statistics = EvolutionStatistics.ofNumber();

        final Phenotype<DoubleGene, Double> best = engine.stream()
            .limit(bySteadyFitness(7))
            .peek(statistics)
            .collect(toBestPhenotype());

        System.out.println(statistics);
        System.out.println(best);
    }
}
```

The console output of the program shows, that **Jenetics** finds the *optimal* solution after 38 generations.

```
+---------------------------------------------------------------------------+
|  Time statistics                                                          |
+---------------------------------------------------------------------------+
|             Selection: sum=0.209185134000 s; mean=0.005504871947 s        |
|              Altering: sum=0.295102044000 s; mean=0.007765843263 s        |
|   Fitness calculation: sum=0.176879937000 s; mean=0.004654735184 s        |
|     Overall execution: sum=0.664517256000 s; mean=0.017487296211 s        |
+---------------------------------------------------------------------------+
|  Evolution statistics                                                     |
+---------------------------------------------------------------------------+
|           Generations: 38                                                 |
|               Altered: sum=7,549; mean=198.657894737                      |
|                Killed: sum=0; mean=0.000000000                            |
|              Invalids: sum=0; mean=0.000000000                            |
+---------------------------------------------------------------------------+
|  Population statistics                                                    |
+---------------------------------------------------------------------------+
|                   Age: max=8; mean=1.100211; var=1.814053                 |
|               Fitness:                                                    |
```

```
|                       min  = 0.000000000000                                |
|                       max  = 63.672604047475                               |
|                       mean = 3.484157452128                                |
|                       var  = 71.047475139018                               |
|                       std  = 8.428966433616                                |
+----------------------------------------------------------------------------+
[[[-1.3226168588424143E-9],[-1.096964971404292E-9]]] --> 0.0
```

9.4 0/1 Knapsack

In the knapsack problem[57] a set of items, together with it's size and value, is given. The task is to select a disjoint subset so that the total size does not exceed the knapsack size. For solving the 0/1 knapsack problem we define a `BitChromosome`, one bit for each item. If the i^{th} bit is set to one the i^{th} item is selected.

```java
import static org.jenetics.engine.EvolutionResult.toBestPhenotype;
import static org.jenetics.engine.limit.bySteadyFitness;

import java.util.Random;
import java.util.function.Function;
import java.util.stream.Collector;
import java.util.stream.Stream;

import org.jenetics.BitGene;
import org.jenetics.Mutator;
import org.jenetics.Phenotype;
import org.jenetics.RouletteWheelSelector;
import org.jenetics.SinglePointCrossover;
import org.jenetics.TournamentSelector;
import org.jenetics.engine.Engine;
import org.jenetics.engine.EvolutionStatistics;
import org.jenetics.engine.codecs;
import org.jenetics.util.ISeq;
import org.jenetics.util.RandomRegistry;

// The main class.
public class Knapsack {

    // This class represents a knapsack item, with a specific
    // "size" and "value".
    final static class Item {
        public final double size;
        public final double value;

        Item(final double size, final double value) {
            this.size = size;
            this.value = value;
        }

        // Create a new random knapsack item.
        static Item random() {
            final Random r = RandomRegistry.getRandom();
            return new Item(
                r.nextDouble()*100,
                r.nextDouble()*100
            );
        }

        // Collector for summing up the knapsack items.
```

[57] https://en.wikipedia.org/wiki/Knapsack_problem

```java
        static Collector<Item, ?, Item> toSum() {
            return Collector.of(
                () -> new double[2],
                (a, b) -> {a[0] += b.size; a[1] += b.value;},
                (a, b) -> {a[0] += b[0]; a[1] += b[1]; return a;},
                r -> new Item(r[0], r[1])
            );
        }
    }

    // Creating the fitness function.
    static Function<ISeq<Item>, Double>
    fitness(final double size) {
        return items -> {
            final Item sum = items.stream().collect(Item.toSum());
            return sum.size <= size ? sum.value : 0;
        };
    }

    public static void main(final String[] args) {
        final int nitems = 15;
        final double kssize = nitems*100.0/3.0;

        final ISeq<Item> items =
            Stream.generate(Item::random)
                .limit(nitems)
                .collect(ISeq.toISeq());

        // Configure and build the evolution engine.
        final Engine<BitGene, Double> engine = Engine
            .builder(fitness(kssize), codecs.ofSubSet(items))
            .populationSize(500)
            .survivorsSelector(new TournamentSelector<>(5))
            .offspringSelector(new RouletteWheelSelector<>())
            .alterers(
                new Mutator<>(0.115),
                new SinglePointCrossover<>(0.16))
            .build();

        // Create evolution statistics consumer.
        final EvolutionStatistics<Double, ?>
            statistics = EvolutionStatistics.ofNumber();

        final Phenotype<BitGene, Double> best = engine.stream()
            // Truncate the evolution stream after 7 "steady"
            // generations.
            .limit(bySteadyFitness(7))
            // The evolution will stop after maximal 100
            // generations.
            .limit(100)
            // Update the evaluation statistics after
            // each generation
            .peek(statistics)
            // Collect (reduce) the evolution stream to
            // its best phenotype.
            .collect(toBestPhenotype());

        System.out.println(statistics);
        System.out.println(best);
    }
}
```

The console out put for the Knapsack GA will look like the listing beneath.

```
+---------------------------------------------------------------------------+
|  Time statistics                                                          |
+---------------------------------------------------------------------------+
|             Selection: sum=0.044465978000 s; mean=0.005558247250 s        |
|              Altering: sum=0.067385211000 s; mean=0.008423151375 s        |
|   Fitness calculation: sum=0.037208189000 s; mean=0.004651023625 s        |
|     Overall execution: sum=0.126468539000 s; mean=0.015808567375 s        |
+---------------------------------------------------------------------------+
|  Evolution statistics                                                     |
+---------------------------------------------------------------------------+
|           Generations: 8                                                  |
|               Altered: sum=4,842; mean=605.250000000                      |
|                Killed: sum=0; mean=0.000000000                            |
|              Invalids: sum=0; mean=0.000000000                            |
+---------------------------------------------------------------------------+
|  Population statistics                                                    |
+---------------------------------------------------------------------------+
|                   Age: max=7; mean=1.387500; var=2.780039                 |
|               Fitness:                                                    |
|                      min  = 0.000000000000                                |
|                      max  = 542.363235999342                              |
|                      mean = 436.098248628661                              |
|                      var  = 11431.801291812390                            |
|                      std  = 106.919601999878                              |
+---------------------------------------------------------------------------+
[01111011|10111101] --> 542.3632359993417
```

9.5 Traveling salesman

The Traveling Salesman problem[58] is one of the classical problems in computational mathematics and it is the most notorious NP-complete problem. The goal is to find the shortest distance, or the path, with the least costs, between N different cities. Testing all possible path for N cities would lead to $N!$ checks to find the shortest one.

The following example uses a path where the cities are lying on a circle. That means, the optimal path will be a polygon. This makes it easier to check the quality of the found solution.

```java
import static java.lang.Math.PI;
import static java.lang.Math.abs;
import static java.lang.Math.sin;
import static org.jenetics.engine.EvolutionResult.toBestPhenotype;
import static org.jenetics.engine.limit.bySteadyFitness;

import java.util.stream.IntStream;

import org.jenetics.EnumGene;
import org.jenetics.Genotype;
import org.jenetics.Optimize;
import org.jenetics.PartiallyMatchedCrossover;
import org.jenetics.PermutationChromosome;
import org.jenetics.Phenotype;
import org.jenetics.SwapMutator;
import org.jenetics.engine.Engine;
import org.jenetics.engine.EvolutionStatistics;

public class TravelingSalesman {

    // Problem initialization:
    // Calculating the adjacence matrix of the "city" distances.
    private static final int STOPS = 20;
    private static final double[][] ADJACENCE = matrix(STOPS);
```

[58] https://en.wikipedia.org/wiki/Travelling_salesman_problem

```java

    private static double[][] matrix(int stops) {
        final double radius = 10.0;
        double[][] matrix = new double[stops][stops];

        for (int i = 0; i < stops; ++i) {
            for (int j = 0; j < stops; ++j) {
                matrix[i][j] = chord(stops, abs(i - j), radius);
            }
        }
        return matrix;
    }

    private static double chord(int stops, int i, double r) {
        return 2.0*r*abs(sin((PI*i)/stops));
    }

    // Calculate the path length of the current genotype.
    private static
    Double dist(final Genotype<EnumGene<Integer>> gt) {
        // Convert the genotype to the traveling path.
        final int[] path = gt.getChromosome().stream()
            .mapToInt(EnumGene<Integer>::getAllele)
            .toArray();

        // Calculate the path distance.
        return IntStream.range(0, STOPS)
            .mapToDouble(i ->
                ADJACENCE[path[i]][path[(i + 1)%STOPS]])
            .sum();
    }

    public static void main(String[] args) {
        final Engine<EnumGene<Integer>, Double> engine = Engine
            .builder(
                TravelingSalesman::dist,
                PermutationChromosome.ofInteger(STOPS))
            .optimize(Optimize.MINIMUM)
            .maximalPhenotypeAge(11)
            .populationSize(500)
            .alterers(
                new SwapMutator<>(0.2),
                new PartiallyMatchedCrossover<>(0.35))
            .build();

        // Create evolution statistics consumer.
        final EvolutionStatistics<Double, ?>
            statistics = EvolutionStatistics.ofNumber();

        final Phenotype<EnumGene<Integer>, Double> best =
            engine.stream()
            // Truncate the evolution stream after 15 "steady"
            // generations.
            .limit(bySteadyFitness(15))
            // The evolution will stop after maximal 250
            // generations.
            .limit(250)
            // Update the evaluation statistics after
            // each generation
            .peek(statistics)
            // Collect (reduce) the evolution stream to
            // its best phenotype.
```

```
                .collect(toBestPhenotype());

        System.out.println(statistics);
        System.out.println(best);
    }
}
```

The Traveling Salesman problem is a very good example which shows you how to solve combinatorial problems with an GA. **Jenetics** contains several classes which will work very well with this kind of problems. Wrapping the base *type* into an `EnumGene` is the first thing to do. In our example, every city has an unique number, that means we are wrapping an `Integer` into an `EnumGene`. Creating a genotype for integer values is very easy with the factory method of the `PermutationChromosome`. For other data types you have to use one of the constructors of the permutation chromosome. As alterers, we are using a swap-mutator and a partially-matched crossover. These alterers guarantees that no invalid solutions are created—every city exists exactly once in the altered chromosomes.

```
+---------------------------------------------------------------------------+
|  Time statistics                                                          |
+---------------------------------------------------------------------------+
|             Selection: sum=0.134312100000 s; mean=0.001618218072 s        |
|              Altering: sum=0.272923323000 s; mean=0.003288232807 s        |
|   Fitness calculation: sum=0.171154575000 s; mean=0.002062103313 s        |
|     Overall execution: sum=0.571970865000 s; mean=0.006891215241 s        |
+---------------------------------------------------------------------------+
|  Evolution statistics                                                     |
+---------------------------------------------------------------------------+
|           Generations: 83                                                 |
|               Altered: sum=117,315; mean=1413.433734940                   |
|                Killed: sum=55; mean=0.662650602                           |
|              Invalids: sum=0; mean=0.000000000                            |
+---------------------------------------------------------------------------+
|  Population statistics                                                    |
+---------------------------------------------------------------------------+
|                   Age: max=11; mean=1.608048; var=4.913384                |
|               Fitness:                                                    |
|                      min  = 95.823941038289                               |
|                      max  = 352.556531948213                              |
|                      mean = 162.422468571595                              |
|                      var  = 3846.044938421069                             |
|                      std  = 62.016489246176                               |
+---------------------------------------------------------------------------+
[12|11|10|1|2|3|4|5|6|7|8|9|0|19|18|17|16|15|14|13] --> 95.82394103828862
```

The listing above shows the output generated by our example. The last line represents the phenotype of the best solution found by the GA, which represents the traveling path. As you can see, the GA has found the shortest path, in reverse order.

9.6 Evolving images

The following example tries to approximate a given image by semitransparent polygons.[59] It comes with an Swing UI, where you can immediately start your own experiments. After compiling the sources with

```
$ ./gradlew jar
```

you can start the example by calling

[59]Original idea by Roger Johansson `http://rogeralsing.com/2008/12/07/genetic-programming-evolution-of-mona-lisa`.

```
$ ./jrun org.jenetics.example.image.EvolvingImages
```

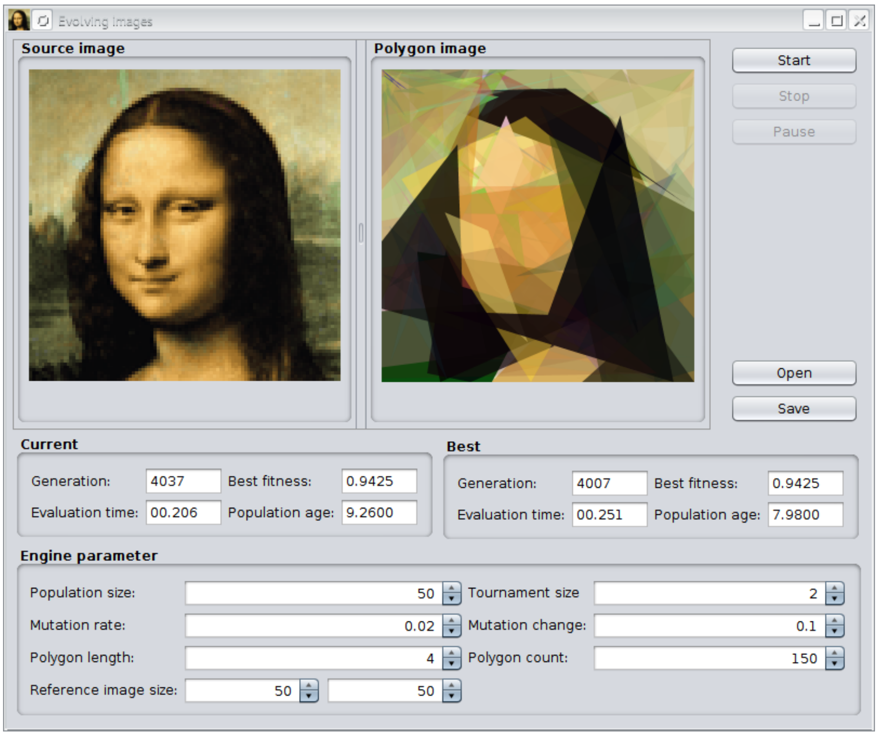


Figure 9.3: Evolving images UI

Figure 9.3 show the GUI after evolving the default image for about 4,000 generations. With the »Open« button it is possible to load other images for *polygonization*. The »Save« button allows to store *polygonized* images in PNG format to disk. At the button of the UI, you can change some of the GA parameters of the example:

Population size The number of individual of the population.

Tournament size The example uses a `TournamentSelector` for selecting the offspring population. This parameter lets you set the number of individual used for the tournament step.

Mutation rate The probability that a polygon *component* (color or vertex position) is altered.

Mutation magnitude In case a polygon *component* is going to be mutated, its value will be randomly modified in the uniform range of $[-m, +m]$.

Polygon length The number of edges (or vertices) of the created polygons.

Polygon count The number of polygons of one individual (`Genotype`).

Reference image size To improve the processing speed, the fitness of a given polygon set (individual) is not calculated with the full sized image. Instead an scaled reference image with the given size is used. A smaller reference image will speed up the calculation, but will also reduce the accuracy.

It is also possible to run and configure the *Evolving Images* example from the command line. This allows to do long running evolution *experiments* and save polygon images every n generations—specified with the `--image-generation` parameter.

```
$ ./jrun org.jenetics.example.image.EvolvingImages evolve \
        --engine-properties engine.properties \
        --input-image monalisa.png \
        --output-dir evolving-images \
        --generations 10000 \
        --image-generation 100
```

Every command line argument has proper default values, so that it is possible to start it without parameters. Listing 35 shows the default values for the GA engine if the `--engine-properties` parameter is not specified.

```
population_size=50
tournament_size=3
mutation_rate=0.025
mutation_multitude=0.15
polygon_length=4
polygon_count=250
reference_image_width=60
reference_image_height=60
```

Listing 35: Default `engine.properties`

For a quick start, you can simply call

```
$ ./jrun org.jenetics.example.image.EvolvingImages evolve
```

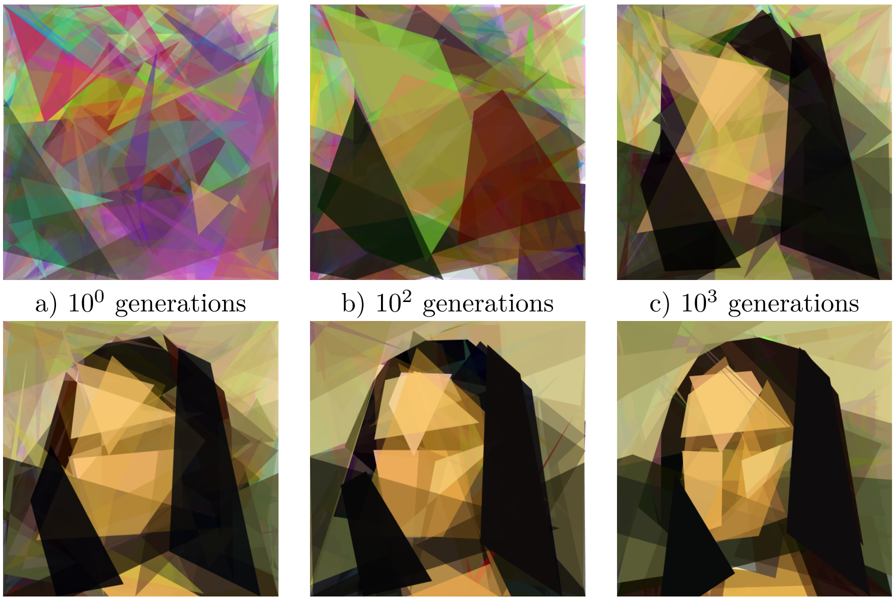
The images in figure 9.4 on the following page shows the resulting polygon images after the given number of generations. They where created with the command line version of the program using the default `engine.properties` file (listing 35):

```
$ ./jrun org.jenetics.example.image.EvolvingImages evolve \
        --generations 1000000 \
        --image-generation 100
```

10 Build

For building the **Jenetics** library from source, download the most recent, stable package version from `https://sourceforge.net/projects/jenetics/files/latest/download` or `https://github.com/jenetics/jenetics/releases` and extract it to some build directory.

```
$ unzip jenetics-<version>.zip -d <builddir>
```



a) 10^0 generations b) 10^2 generations c) 10^3 generations

d) 10^4 generations e) 10^5 generations f) 10^6 generations

Figure 9.4: Evolving *Mona Lisa* images

`<version>` denotes the actual **Jenetics** version and `<builddir>` the actual build directory. Alternatively you can check out the latest version from the Git `master` branch.

```
$ git clone https://github.com/jenetics/jenetics.git \
          <builddir>
```

Jenetics uses Gradle[60] as build system and organizes the source into *sub*-projects (*modules*).[61] Each *sub*-project is located in it's own *sub*-directory:

- **`org.jenetics`**: This project contains the source code and tests for the **Jenetics** *core*-module.
- **`org.jenetics.example`**: This project contains example code for the *ore*-module.
- **`org.jenetics.doc`**: Contains the *code* of the web-site and *this* manual.

For building the library change into the `<builddir>` directory (or one of the *module* directory) and call one of the available *tasks*:

- **`compileJava`**: Compiles the **Jenetics** sources and copies the class files to the `<builddir>/<module-dir>/build/classes/main` directory.
- **`jar`**: Compiles the sources and creates the JAR files. The artifacts are copied to the `<builddir>/<module-dir>/build/libs` directory.

[60] `http://gradle.org/downloads`

[61] If you are calling the `gradlew` script (instead of `gradle`), which are part of the downloaded package, the proper Gradle version is automatically downloaded and you don't have to install Gradle explicitly.

- **`test`**: Compiles and executes the unit tests. The test results are printed onto the console and a test-report, created by *TestNG*, is written to `<builddir>/<module-dir>` directory.
- **`javadoc`**: Generates the API documentation. The Javadoc is stored in the `<builddir>/<module-dir>/build/docs` directory
- **`clean`**: Deletes the `<builddir>/build/*` directories and removes all generated artifacts.

For `building the library from` the source, call

```
$ cd <build-dir>
$ gradle jar
```

or

```
$ ./gradlew jar
```

if you don't have the the Gradle build system installed—calling the the Gradle wrapper script will download all needed files and trigger the build task afterwards.

IDE integration Gradle has tasks which creates the project file for Eclipse[62] and IntelliJ IDEA[63]. Call

```
$ ./gradlew <eclipse|idea>
```

for creating the project files for Eclipse or IntelliJ, respectively.

External library dependencies The following external projects are used for running and/or building the **Jenetics** library.

- ***TestNG***
  - **Version**: *6.11*
  - **Homepage**: *http://testng.org/doc/index.html*
  - **License**: *Apache License, Version 2.0*
  - **Scope**: *test*
- ***Apache Commons Math***
  - **Version**: *3.6.1*
  - **Homepage**: *http://commons.apache.org/proper/commons-math/*
  - **Download**: *http://tweedo.com/mirror/apache/commons/math/binaries/commons-math3-3.6.1-bin.zip*
  - **License**: *Apache License, Version 2.0*
  - **Scope**: *test*

[62] http://www.eclipse.org/
[63] http://www.jetbrains.com/idea/

- ***Java2Html***
  - **Version**: *5.0*
  - **Homepage**: *http://www.java2html.de/*
  - **Download**: *http://www.java2html.de/java2html_50.zip*
  - **License**: *GPL or CPL1.0*
  - **Scope**: *javadoc*
- ***Gradle***
  - **Version**: 3.5
  - **Homepage**: *http://gradle.org/*
  - **Download**: *http://services.gradle.org/distributions/gradle-3.5-bin.zip*
  - **License**: *Apache License, Version 2.0*
  - **Scope**: *build*

Maven Central The whole **Jenetics** package can also be downloaded from the *Maven Central* repository `http://repo.maven.apache.org/maven2`:

`pom.xml` snippet for Maven

```
<dependency>
    <groupId>io.jenetics</groupId>
    <artifactId>jenetics</artifactId>
    <version>3.8.0</version>
</dependency>
```

Gradle

```
'io.jenetics:jenetics:3.8.0'
```

11 License

The library itself is licensed under the Apache License, Version 2.0.

References

[1] Thomas Back. *Evolutionary Algorithms in Theory and Practice.* Oxford Univiversity Press, 1996.

[2] James E. Baker. Reducing bias and inefficiency in the selection algorithm. *Proceedings of the Second International Conference on Genetic Algorithms and their Application*, pages 14–21, 1987.

[3] Shumeet Baluja and Rich Caruana. Removing the genetics from the standard genetic algorithm. pages 38–46. Morgan Kaufmann Publishers, 1995.

[4] Heiko Bauke. Tina's random number generator library. *http://numbercrunch.de/trng/trng.pdf*, 2011.

[5] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4:361–394, 1997.

[6] P.K. Chawdhry, R. Roy, and R.K. Pant. *Soft Computing in Engineering Design and Manufacturing.* Springer London, 1998.

[7] Richard Dawkins. *The Blind Watchmaker.* New York: W. W. Norton & Company, 1986.

[8] Sean Luke. *Essentials of Metaheuristics.* Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[9] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution.* Springer, 1996.

[10] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, USA, 1998.

[11] Heinz Mühlenbein and Dirk Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm i. continuous parameter optimization. 1(1):25–49.

[12] Albert Nijenhuis and Herbert Wilf. *Combinatorial Algorithms for Computers and Calculators.* Academic Press, second edition, 1978.

[13] Oracle. Value-based classes. *https://docs.oracle.com/javase/8/docs/api/-java/lang/doc-files/ValueBased.html*, 2014.

[14] Charles C. Palmer and Aaron Kershenbaum. An approach to a problem in network design using genetic algorithms. *Networks*, 26(3):151–163, 1995.

[15] Franz Rothlauf. *Representations for Genetic and Evolutionary Algorithms.* Springer, 2 edition, 2006.

[16] Daniel Shiffman. *The Nature of Code.* The Nature of Code, 1 edition, 12 2012.

[17] S. N. Sivanandam and S. N. Deepa. *Introduction to Genetic Algorithms.* Springer, 2010.

[18] W. Vent. Rechenberg, ingo, evolutionsstrategie — optimierung technischer systeme nach prinzipien der biologischen evolution. 170 s. mit 36 abb. frommann-holzboog-verlag. stuttgart 1973. broschiert. *Feddes Repertorium*, 86(5):337–337, 1975.

[19] Eric W. Weisstein. Scalar function. *http://mathworld.wolfram.com/-ScalarFunction.html*, 2015.

[20] Eric W. Weisstein. Vector function. *http://mathworld.wolfram.com/-VectorFunction.html*, 2015.

[21] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

[22] Mark Wolters. A genetic algorithm for selection of fixed-size subsets with application to design problems. *Journal of Statistical Software*, 68(1):1–18, 2015.

Index