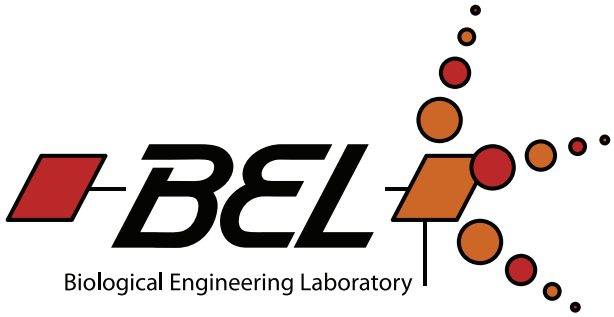
BEL–Genetic Algorithm
A .NET Computational Package Written in Zonnon
Part II of the Users Guide

Alan D. Freed

Clifford H. Spicer Chair in Engineering
College of Science, Engineering & Technology
Saginaw Valley State University
202 Pioneer Hall, 7400 Bay Road
University Center, MI 48710
E-Mail: adfreed@svsu.edu

October 23, 2013



Abstract

This document describes an application library for the BEL framework. BEL is a software library written in Zonnon for the .NET and Mono frameworks. The application addressed here is that of a Genetic Algorithm (GA) developed for the purpose of acquiring parameter estimates belonging to any user-definable mathematical model that engineers, scientists, and the like would want to use to describe experimental data or observations through models.

1 Introduction

BEL is an acronym taken for my research laboratory: The Biological Engineering Laboratory at Saginaw Valley State University. What BEL is and what it provides are quite different from what one might expect, given this affiliation. BEL arose out of the author's desire to interface with the .NET and Mono Frameworks for the purpose of writing computational software programs in the Zonnon programming language [1].

Zonnon is the most recent descendant from the family tree Euler/Algol/Pascal/Modula/Oberon of programming languages developed by Profs. Niklaus Wirth and Jürg Gutknecht from the Computer Systems Institute at ETH Zurich (Eidgenössische Technische Hochschule, Zentrum, i.e., the Swiss Federal Institute of Technology, Zurich). Zonnon was created specifically for .NET, with compiler versions available for both the .NET and Mono Frameworks. Sponsored by Xamarin, Mono is an open source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime. The Zonnon compiler targets the .NET Framework 2.0.

The Zonnon compiler, documentation, example programs, and even BEL itself, are free to download from `http://www.zonnon.ethz.ch`. The home page for Microsoft's .NET Framework is at `http:/`

/www.microsoft.com/net. Mono's is at http://www.mono-project.com/Main_Page.

Licensing of the BEL library, both its software and documentation, is addressed in the *Licensing of the Software and its Documentation* part to this user guide.

1.1 Version

This document corresponds to software version 4.2 of BEL-GAlg, October 23, 2013. Zonnon targets the .NET Framework 2.

1.2 Required Libraries

The only external library that is required to compile BEL-GAlg is the one that you created when you built BEL.

1.3 Known Issues

None

1.4 Feedback

I consider this to be a living document. If there is some aspect that is wanting, or if you have corrections and/or additions that you believe would benefit other users of BEL-GAlg, please forward them to adfreed@svsu.edu.

2 Overview

GOLDBERG [2] tells his readers what genetic algorithms (GAs) are, conceptually, in the opening paragraph of his book:

> "Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search. In every generation, a new set of artificial creatures (strings) is created using bits and pieces of the fittest of the old; an occasional new part is tried for good measure. While randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search points with expected improved performance."

GAs have in common a population of individuals, a means to determine the fitness of each individual within the population, the pairing of individuals for reproduction according to their fitness, the cross-fertilization of genetic material from parents to produce offspring, and a random chance of a mutation occurring in the off-springs' genetic code. A new generation replaces the existing one, and the reproduction cycle starts all over again. This process is repeated to convergence, i.e., the attainment of a uniform population of the most fit creature.

Seven modules (presented from the bottom up) were written to implement a GA for the purpose of obtaining parameter estimates in functions used to describe observations or experimental data. Their software interfaces are presented in App. B.

The genetic algorithm that is distributed with BEL-GAlg draws heavily on the algorithms and Pascal code of GOLDBERG [2, 3]. His books focus on what is called a simple genetic algorithm (SGA), where the probabilities of mutation and crossover are fixed over the lifetime of the colony. This software implements an adaptive genetic algorithm wherein these probabilities vary over the solution space [4, pp. 121-122]. SCHMITT [5] has shown that GAs can be derived from the principle of maximum entropy governing a Markovian process. It turns out that GAs are a great application to showcase the capabilities of the Zonnon programming language.

GAs are but one of many optimization techniques that exist. There is a theorem in the optimization literature called the **no free lunch** theorem. It states that the overall performance of any optimization algorithm, when evaluated over the set of all possible optimization problems, is no

different than any other optimization technique. An apparent advantage of one algorithm over another resides strictly with its application and, quite often, the personal taste of the user. It has been the author's experience that GAs work very well for parameter estimations of non-linear material models.

3 Statistics

GAs are designed to maximize a fitness pertinent to a species of *creatures*. How one defines 'fitness' is left for the user to specify. The second moment from sample statistics is used here as the measure for fitness. An optimal solution is one that minimizes the noise between experimental data and the model chosen to fit them. In terms of statistics, this is equivalent to saying that we want to minimize a variance describing the difference between model and experiment. GAs choose objective functions that maximize fitness, so in our application we will select an objective function that is the reciprocal of such a variance, i.e., one arising from the differences between experimental data and their predictions from a model of correlation.

3.1 Nomenclature

C	number of control variables
N	number of observations used in optimization
O	total number of experimental observations
P	number of model parameters to be estimated
R	number of response variables

3.2 Random Variables

There are considered to be N, experimental, data vectors $\mathbb{X}_n$, $n = 1, 2, \ldots, N$, that are aggregated as columns into a matrix $\mathbb{X} = [\![\mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_N]\!]$ wherein each vector $\mathbb{X}_n$ has R row elements, i.e., $\mathbb{X}_n = \mathbb{X}(t_n) = \{x_{1n}, x_{2n}, \ldots, x_{Rn}\}^T$ evaluated at an instance t_n, where R is the number of responses or dependent variables that have been measured, and N is the number of experimental measurements made. From these data, estimates for P parameters $\theta = \{\theta_1, \theta_2, \ldots, \theta_P\}^T$ that belong to some model $\Xi(t, \theta)$ are to be acquired. A call to procedure `Theory` returns an $R \times N$ matrix containing its predictions $\Xi(t_n, \theta) = [\![\mathbb{\Xi}_1(\theta), \mathbb{\Xi}_2(\theta), \ldots, \mathbb{\Xi}_N(\theta)]\!]$, wherein each column vector $\mathbb{\Xi}_n$ has R row elements, i.e., $\mathbb{\Xi}_n(\theta) = \Xi(t_n, \theta) = \{\xi_{1n}(\theta), \xi_{2n}(\theta), \ldots, \xi_{Rn}(\theta)\}^T$ that have been derived using parameters θ. In what follows, the θ dependence of the model $\Xi(t_n, \theta)$, its solution vectors $\mathbb{\Xi}_n(\theta)$, and their components $\xi_{rn}(\theta)$ are not explicitly written out in an effort to keep the notation more compact. This dependence is, however, tacitly understood to exist.

3.2.1 $R = 1$

We begin our discussion by considering a single response variable, i.e., let $R = 1$. For this condition we introduce a random variable to represent the residual error between two other random variables: one for the experiment, the other for a model, viz.,

$$\mathbb{e} = a\big(\mathbb{X} - \mathbb{\Xi}\big),$$

whose components are $\mathbb{e} = \{e_1, e_2, \ldots, e_N\}^T$. To help one acquire a sense of intuition after running multiple optimizations, it is instructive to introduce a scaling factor a such that

$$a = \frac{1}{\max_{n=1}^{N}(|x_n|)},$$

which places all experimental data within the unit interval $[-1, 1]$. Ideally, one would want the data $a\mathbb{X}$ to be uniformly dense over this interval. If this is not the case with a particular data set of interest, it may be beneficial to transform it first, e.g., by taking its logarithm. Of course, this implies that one would have to introduce this same transformation scheme into one's coding of procedure `Theory` that is being used to model these data. *Parameter estimation is part science and part art.*

The expected value for this residual error is

$$\mathrm{E}(\mathbb{e}) = a\mathrm{E}(\mathbb{X}) - a\mathrm{E}(\mathbb{\Xi}),$$

where sample statistics are used to establish the expectations residing on the right-hand side of this

equation, viz.,

$$\mathrm{E}(\mathbb{X}) = \frac{1}{N}\sum_{n=1}^{N} x_n,$$
$$\mathrm{E}(\mathbb{\Xi}) = \frac{1}{N}\sum_{n=1}^{N} \xi_n.$$

From its definition of $\mathrm{VAR}(\mathbb{e}) = \mathrm{E}([\mathbb{e} - \mathrm{E}(\mathbb{e})]^2)$, it follows that the variance of this residual error can be described in terms of the variances and the covariance of its two, dependent, random variables; specifically,

$$\mathrm{VAR}(\mathbb{e}) = a^2\big(\mathrm{VAR}(\mathbb{X})+\mathrm{VAR}(\mathbb{\Xi})-2\,\mathrm{COV}(\mathbb{X},\mathbb{\Xi})\big),$$

where the terms on the right-hand side are given by

$$\mathrm{VAR}(\mathbb{X}) = \mathrm{E}(\mathbb{X}^2) - \big(\mathrm{E}(\mathbb{X})\big)^2,$$
$$\mathrm{VAR}(\mathbb{\Xi}) = \mathrm{E}(\mathbb{\Xi}^2) - \big(\mathrm{E}(\mathbb{\Xi})\big)^2,$$
$$\mathrm{COV}(\mathbb{X},\mathbb{\Xi}) = \mathrm{E}(\mathbb{X}\,\mathbb{\Xi}) - \mathrm{E}(\mathbb{X})\,\mathrm{E}(\mathbb{\Xi}).$$

Sample statistics quantifies the second-order moments, too, viz.

$$\mathrm{E}(\mathbb{X}^2) = \frac{1}{N}\sum_{n=1}^{N} x_n^2$$
$$\mathrm{E}(\mathbb{\Xi}^2) = \frac{1}{N}\sum_{n=1}^{N} \xi_n^2$$
$$\mathrm{E}(\mathbb{X}\,\mathbb{\Xi}) = \frac{1}{N}\sum_{n=1}^{N} x_n\,\xi_n$$

An admissible objective function $\Phi(\mathbb{\theta})$ that can be maximized by a GA (or like algorithm) is given by

$$\Phi = \frac{\max_{n=1}^{N} x_n^2}{\mathrm{VAR}(\mathbb{X}) + \mathrm{VAR}(\mathbb{\Xi}) - 2\,\mathrm{COV}(\mathbb{X},\mathbb{\Xi})}.$$

This objective function applies for a single response variable. It will be at a maximum whenever its denominator is at a minimum, which will occur whenever the covariance between model and experiment is at its maximum.

Covariance is neglected in almost all optimization strategies, but here it plays the central role of the process. *Effectively, the optimization strategy is to maximize the covariance between model and experiment.*

3.2.2 $R = 2$

In this scenario we redefine our random variable $\mathbb{e}$ as $\mathbb{\varepsilon}$ to distinguish the sum of two residual errors $\mathbb{\varepsilon}$ from its two distinct errors $\mathbb{e}_1$ and $\mathbb{e}_2$ as defined in

$$\mathbb{\varepsilon} = \tfrac{1}{2}\big(\mathbb{e}_1 + \mathbb{e}_2\big),$$

where, like before,

$$\mathbb{e}_r = a_r\big(\mathbb{X}_r - \mathbb{\Xi}_r\big), \quad r = 1, 2,$$

with

$$a_r = \frac{1}{\max_{n=1}^{N}(|x_{rn}|)}, \quad r = 1, 2,$$

such that index $r = 1$ designates the first response variable, and $r = 2$ the second.

In this case, the expected value for $\mathbb{\varepsilon}$ is

$$\mathrm{E}(\mathbb{\varepsilon}) = \tfrac{1}{2}\big(\mathrm{E}(\mathbb{e}_1) + \mathrm{E}(\mathbb{e}_2)\big),$$

wherein

$$\mathrm{E}(\mathbb{e}_r) = a_r\big(\mathrm{E}(\mathbb{X}_r) - \mathrm{E}(\mathbb{\Xi}_r)\big),$$

with the expectations on the right-hand side of this formula being evaluated via sample statistics, just as they were defined in the previous section. Likewise, the variance of $\mathbb{\varepsilon}$ is found to be

$$\mathrm{VAR}(\mathbb{\varepsilon}) = \tfrac{1}{4}\big(\mathrm{VAR}(\mathbb{e}_1) + \mathrm{VAR}(\mathbb{e}_2) + 2\,\mathrm{COV}(\mathbb{e}_1,\mathbb{e}_2)\big),$$

where here the contribution from the covariance is *added* to that of its two variances, instead of being *subtracted* as before. The variances on the right-hand side of this formula, on the other hand, are described by

$$\mathrm{VAR}(\mathbb{e}_r) = a_r^2\big(\mathrm{VAR}(\mathbb{X}_r)+\mathrm{VAR}(\mathbb{\Xi}_r)-2\,\mathrm{COV}(\mathbb{X}_r,\mathbb{\Xi}_r)\big).$$

These variances and covariance are evaluated with sample statistics in the same manner as their counterparts were in the prior section.

Whenever the two residual errors $\mathbb{e}_1$ and $\mathbb{e}_2$ act independent of one another, then, by definition, their covariance will be zero, viz., $\mathrm{COV}(\mathbb{e}_1,\mathbb{e}_2) = 0$, which significantly simplifies

one's calculation for the total system variance in that now

$$\mathrm{VAR}(\mathscr{E}) = \tfrac{1}{4}\big(\mathrm{VAR}(\mathscr{e}_1) + \mathrm{VAR}(\mathscr{e}_2)\big).$$

This simplifying assumption is adopted in this release of the BEL-GAlg software.

It is important to emphasize that what is being assumed here is: The *residual errors* $\mathscr{e}_1$ and $\mathscr{e}_2$ are considered to be independent random variables. This does not imply that their *response variables* $\mathbb{X}_1$ and $\mathbb{X}_2$ are (or are not) independent of one another. That is a separate issue.

3.2.3 Arbitrary R

In a general setting of R separate response variables, one constructs an overall residual error as a weighted average over all of the individual residual errors such that

$$\mathscr{E} = \frac{1}{R}\sum_{r=1}^{R} \mathscr{e}_r,$$

wherein

$$\mathscr{e}_r = a_r\big(\mathbb{X}_r - \bar{\mathbb{X}}_r\big),$$

with normalization

$$a_r = \frac{1}{\max_{n=1}^{N}(|x_{rn}|)}.$$

The expectation of this collective residual error is

$$\mathrm{E}(\mathscr{E}) = \frac{1}{R}\sum_{r=1}^{R} \mathrm{E}(\mathscr{e}_r),$$

wherein

$$\mathrm{E}(\mathscr{e}_r) = a_r\big(\mathrm{E}(\mathbb{X}_r) - \mathrm{E}(\bar{\mathbb{X}}_r)\big),$$

while its variance is considered to be described by

$$\mathrm{VAR}(\mathscr{E}) = \frac{1}{R^2}\sum_{r=1}^{R} \mathrm{VAR}(\mathscr{e}_r),$$

wherein

$$\mathrm{VAR}(\mathscr{e}_r) = a_r^2\big(\mathrm{VAR}(\mathbb{X}_r)+\mathrm{VAR}(\bar{\mathbb{X}}_r)-2\,\mathrm{COV}(\mathbb{X}_r, \bar{\mathbb{X}}_r)\big),$$

with the individual r statistics being quantified as in the $R = 1$ case, but now for each r in R. With this information in hand, one can propose an objective function for maximization of the form

$$\Phi = \frac{1}{\mathrm{VAR}(\mathscr{E})},$$

and our statistical foundation is in place. A call to `BelGAlg.Statistics.Fitness` will return this value of Φ for a creature with parameters θ.

3.3 Coefficient of Determination

Objective functions are powerful tools in applications, but they can be difficult to acquire an intuitive feel for 'goodness of fit'. The coefficient of determination is not unique when extended to non-linear optimization algorithms, and many methods for computing such an $\mathcal{R}^2$ value have been proposed over the years. An adjusted or bias corrected coefficient of determination taken from multiple regression analysis has been applied here; specifically,

$$\mathcal{R}^2 = 1 - \frac{N-1}{N-P}\,\frac{\mathrm{E}(\mathscr{E}^2)}{\mathrm{VAR}(\mathbb{X})},$$

where

$$\mathrm{VAR}(\mathbb{X}) = \frac{1}{R^2}\sum_{r=1}^{R} a_r^2\mathrm{VAR}(\mathbb{X}_r),$$

and

$$\mathrm{E}(\mathscr{E}^2) = \frac{1}{R^2}\sum_{r=1}^{R} \mathrm{E}(\mathscr{e}_r^2),$$

wherein

$$\mathrm{E}(\mathscr{e}_r^2) = a_r^2\big(\mathrm{E}(\mathbb{X}_r^2) + \mathrm{E}(\bar{\mathbb{X}}_r^2) - 2\,\mathrm{E}(\mathbb{X}_r\bar{\mathbb{X}}_r)\big).$$

As with the variance of residual error $\mathrm{VAR}(\mathscr{E})$, it is assumed that the squared errors of the residuals $\mathscr{e}_1^2, \mathscr{e}_2^2, \ldots, \mathscr{e}_R^2$ are independent of one another, too, and therefore there expectations can be summed. We recall that $\mathrm{VAR}(\mathscr{E}) = \mathrm{E}(\mathscr{E}^2) - (\mathrm{E}(\mathscr{E}))^2$ and $\mathrm{VAR}(\mathscr{e}_r) = \mathrm{E}(\mathscr{e}_r^2) - (\mathrm{E}(\mathscr{e}_r))^2\ \forall\ r$, where the supposition is that like terms sum over R. Here P is the

dimension of θ, i.e., the number of parameters that are fit, R is the number of response variables being modeled, and N is the number of datum points being fit against. Procedure `RSquared` in module `BelGAlg.Statistics` returns this adjusted $\mathcal{R}^2$ statistic for a given set of parameters θ. The parameters that maximize the fitness function Φ are typically distinct from those that maximize the $\mathcal{R}^2$ statistic, although they are similar.

It bears repeating a phrase that commonly appears whenever an $\mathcal{R}^2$ statistic is being introduced: "Correlation does not imply causation." In other words, good correlation (a high $\mathcal{R}^2$ statistic) must not be used to infer or suggest the existence of a causal relationship between a model and data, i.e., it does not prove that the model is 'correct'.

3.4 The Software

Module `BelGAlg.Statistics` exports four immutable constants: the number of data sets being fit against N as `dimN`, the number of response or independent variables R in the experiment/model as `dimR`, the number of control or dependent variables `dimC` (which must equal `dimR` whenever the data are to be decimated), and the number of parameters allowed to vary P that appear within a model as `dimP`. It also exports two probabilistic functions that are used throughout the GA modules: a biased coin flip via `FlipHeads`, and the return of a `RandomIntergerBetween` two distinct integers.

`Statistics` also exports a procedure type. Instances of `Model` can be submitted to the GA for parameter optimization. They get passed to either procedure `Optimize` or `EasyOptimize` in module `BelGAlg.GeneticAlgorithm`. Type `Model` requires two arguments. The first is a Bel.Types.RealVector of length P that contains the parameters to be estimated. The second is a Bel.Types.RealMatrix with C rows (the number of control variables) and O columns (the total number of experimental observations). A call to a `Model` implementation will return a Bel.Types.RealMatrix in R rows, which is the number of response variables, spanning over the O columns. It contains the predicted response of the model when subjected to the supplied control variables for the given set of model parameters.

Procedure `Configure` allows the user to supply experimental data, and a numerical model whose parameters one hopes to estimate. This procedure has five arguments, the first two of which contain experimental data. The first is a $C \times O$ matrix that holds the control variables to be forwarded to the user-supplied model. The second is a $R \times O$ matrix that contains the experimental output variables, the latter of which the optimizer will attempt to replicate with the model by systematically adjusting its parameters.

The third argument of `Configure` indicates how many of the data points are to be used for optimization. Optimization is an expensive process, and it is often useful to decimate larger data sets into smaller more manageable ones. A decimation scheme similar to that of Doehring *et al.* [6] has been implemented into `Configure`, where the input/output curves are segmented into intervals of like arc length. **Note**: Decimation requires the input and output matrices to have the same dimensions.

The fourth argument supplies the numerical model to the optimizer, while the fifth selects which model parameters are to be allowed to vary. It may be that you know a parameter *a priori* so there is no need to allow the optimizer to seek its value.

Procedure `DataFitAgainst` returns the decimated experimental data set used for parameter estimation. The two arguments are matrices. The first has dimension $C \times N$ while the second has dimension $R \times N$, where C is the number of control variables, R is the number of response variables, and N is the actual number of experimental data points used during an optimization, i.e., $N \leq O$. The first argument contains the experimental inputs. The second argument contains the experimental outputs. They are subsets of the first two arguments supplied to `Configure`. This information may be useful to the user who may want them, e.g., to make graphs of a model's capability to correlate these data.

For these data that are to be fit against, procedure `Theory` returns their corresponding theoretical predictions using the model assigned previously to the GA via procedure `Configure`. `Theory`

requires that the caller send the set of model parameters that are to be used by the theory to provide its predictions, which are returned as a $R \times N$ matrix.

4 Genes

A gene is an information bit in a memory bank that belongs to the overall cache of biological memory that resides within a living organism, often referred to in the GA literature as a *creature*.

Module `BelGAlg.Genes` exports type `Gene`, which implements definitions `Bel.Entity` and `Bel.Typeset`.

The implementation of `Initialize` applies even-odds probability to assign a random value for its allele (the expression of that gene). Allele are randomly assigned in the beginning, i.e., during the procreation, which is where the first colony of individuals originates from, and whenever an individual immigrates into the colony. In later colonies, the allele are assigned through chromosome splitting (occurs with high probability) with the possibility of an additional gene mutation (occurs with small probability), both of which take place at the time of reproduction.

Method `Clone` produces a duplicate of its genetic material that from a biological perspective makes it the appropriate terminology to use to describe its action. In computer science, however, a clone would have the same type, but it would be an empty shell of the original, minus any substantive data, which is how a `Bel.Object.Clone` is typically implemented. (`Gene` does not implement the `Bel.Object` interface.)

`Parse` and `Print` are used to assign and extract the allele of a gene after its initial creation. This is done with strings. A biallelic gene is implemented which has two expressive states: dominant and recessive. Dominant genes are expressed with a 1, while recessive genes are expressed with a 0.[1]

Methods `Pop` and `Put` are for internal use in that they are called by the operator overloading assignment; specifically:

```
operator ":=" (var l : Gene; r : Gene);
begin
   l.Put(r.Pop())
end ":=";
```

This technique allows one to make deep copies of an object via assignment without utilizing the assignment operator := in its implementation.

Method `Mutate` performs a random mutation on a gene, switching its value at some specified probability for mutation. If a gene is mutated, it increments a counter that is used to keep track of the number of mutations that have occurred for output into the final report, cf. App. A.

Method `Equals` tests two genes to determine if they are equivalent. It is called by the overloaded operators "=" and "#".

4.1 Probability of Mutation

In simple genetic algorithms (SGA), like the one developed in the book of Goldberg [2], the probability of a mutation occurring is held fixed over the life of a colony of creatures. An adaptive genetic algorithm [4, pg. 122] is employed in Bel-GAlg. After a generation has been fully populated and the fitness of all creatures in that generation are known, an adaptive probability of mutation P_m is then assigned to each creature according to the formula (which has a typo in [4])

$$P_m = \begin{cases} P^m_{\max} - \frac{(P^m_{\max} - P^m_{\min})(f - f_{\text{avg}})}{f_{\max} - f_{\text{avg}}} & f \geq f_{\text{avg}}, \\ P^m_{\max} & f < f_{\text{avg}}, \end{cases}$$

where $P^m_{\max}$ and $P^m_{\min}$ are the maximum and minimum probabilities for mutation, which are exported from module `BelGAlg.Creatures` through the pair of constants `maxProbabilityOfMutation` and `minProbabilityOfMutation`, where they are assigned values of 0.1 and 0.001, respectively. The fitness of an individual creature is denoted as f,

1. In versions 1.2.1 and earlier, a triallelic gene was also implemented. It had three expressive states: dominant, dominant/recessive and recessive, where a dominant gene was expressed with a 1, a dominant/recessive gene was expressed with a %, and a recessive gene was expressed with a 0. Experimentation with this option found that it brought little to no added value to the overall optimization process, so it was removed with the release of version 2.0 of the software.

while f_{avg} and f_{max} correspond to the average fitness of that generation's population and the fitness of its *elite* creature. This formula is akin to the annealing protocol derived by SCHMITT [7].

At the time of reproduction, the probability of mutation applied to an offspring is the average of the probabilities of mutation belonging to its parents.

5 Chromosomes

A chromosome is a bank of memory comprised of a string of genes that is part of the overall cache of biological memory that resides within a living organism. In genetic algorithms, a creature's chromosome is a register or memory cache for a parameter whose value is being sought via the GA.

In this implementation of a GA, each chromosome is paired with a single model parameter that is being sought through optimization. If a model has five unknown parameters that are to be estimated, for example, then its implementation in BEL-GAlg will result in five unique chromosomes. In this respect, BEL-GAlg differs from GOLDBERG's SGA [2]. His SGA has just one chromosome, whose genotypes for all of the parameters are spliced into a long single strand of genes.

Module `BelGAlg.Chromosomes` exports type `Chromosome`, which is an implementation of the `[]`, `Bel.Entity`, and `Bel.Typeset` definitions. `Chromosome` is a wrapper for the other exported type: `Haploid`, which is an **array * of** `BelGAlg.Genes.Gene`.[2] Type `Chromosome` exports seven methods, in addition to the four that are required by the three definitions being implemented.

Method `Clone` returns an exact duplicate of itself, i.e., a deep copy. `Chromosome` assignment via := yields a shallow copy. To get a deep copy one must call `Clone`. (`Chromosome` does not implement the `Bel.Object` interface.)

Method `Create` receives the lower and upper boundaries that establish the regions of search associated with each parameter, each being assigned to their own chromosome. Also supplied is the number of significant figures in accuracy that one would hope to have in the final result (the author used 4 in the following example). These three factors establish the number of genes required by each chromosome, whose collective sum (over all chromosomes) is exported through method `Length`.

Method `Equals` checks to see if the strings of gene expression between the two chromosomes are the same. `Equals` is called by the overloaded operators "=" and "#".

Method `Mutate` forwards its call to all genes within the chromosome string, where the actual mutations take place.

`Encode` and `Decode` are analogs to `Parse` and `Print`, except they import and export the *phenotype* (i.e., the model parameter) held by the chromosome; whereas, `Parse` and `Print` import and export the *genotype* (the values of their allele). `Encode` and `Decode` provide the mapping functions that bridge a model (whose parameters are to be estimated) with the GA that will estimate them.

Gene expression is GRAY encoded in our GA [8]. GRAY numbers are a binary-like representation whose neighboring values differ by one bit only, which is not true of binary numbers. GRAY code is mapped into binary code. Binary code is mapped into integers. And integers are then converted into their corresponding real phenotype in `Decode`. `Encode` runs these maps in reverse.

E.g., a `BelGAlg.Chromosomes.Chromosome` could have a genotype that looks like

1 0 0 1 0 1 1 1

that, in this case, is a string of eight biallelic genes. The `Get` and `Set` procedures (called by its indexer via the `[]` interface) can access the individual genes in a chromosome, and accept indexes between 0 and `Length` − 1.

A chromosome has two types of riders that position differently. An 'index' rider points directly to a gene, and is used for mutation. Its

2. In versions 1.2.1 and earlier, a diploid chromosome, which is a string of triallelic genes, was also implemented. Experimentation with this option found that it too brought little additional value to the overall optimization process, so it was removed with the release of version 2.0 of the software.

value resides within the interval $[0, \texttt{Length} - 1]$. A 'locus' rider points in between two genes, and is used for crossover (chromosome splitting). Its value belongs to the interval $[1, \texttt{Length} - 1]$. The overloaded indexer for a BEL-GAlg chromosome associates with a chromosome's 'index' rider.

Module `BelGAlg.Chromosomes` also exports the procedure `Crossover`, which is called by higher-level modules to mate two individuals and return their offspring. This is where crossover (the splitting and splicing of chromosomes at reproduction) actually occurs. The first argument establishes the probability for crossover to occur. The second and third arguments provide the happy parents. The fourth argument is a counter that gets incremented if crossover takes place; it is a statistic provided in the final report. The final pair of arguments are the offspring of this mating, which will be clones of their parents if no crossover takes place.

`Crossover` implements a one-point crossover procedure [4]. As an example, consider two parents whose chromosomes are of length six. From a call to a random number generator it is determined that a splice is to take place between the second and third genes, e.g., and therefore

$$\begin{array}{cccccc} 1 & 0 & | & 1 & 1 & 0 & 0 \\ 0 & 1 & | & 0 & 1 & 0 & 1 \end{array} \quad \longrightarrow \quad \begin{array}{cccccc} 1 & 0 & | & 0 & 1 & 0 & 1 \\ 0 & 1 & | & 1 & 1 & 0 & 0 \end{array},$$

where the parents on the left produce the children on the right in this example.

Schemata are the building blocks of genetic algorithms; they are what make them work [2, 3]. A schema can be any string of genes that is shorter than a chromosome; it is a subset to a chromosome. In a genetic algorithm, high-performance, low-order schemata receive exponentially increasing numbers of trials in successive generations, which leads to the fundamental theorem of genetic algorithms [2, 4]. This is why they outperform other stochastic methods, like Monti Carlo techniques. In schemata, some of the genes take on fixed expressions, while others are allowed to have any expression, and are denoted with a $\star$. For example, a schema might look like $1 \ \star \ 0$ of which instances $1\ 0\ 0$ and $1\ 1\ 0$ would both belong. In this example, the schema has dimension three. In the example that follows, schemata up to dimension 6 were used.

5.1 Probability of Crossover

In SGAs, the probability of a crossover occurring within any given chromosome during reproduction is held fixed over the life of a colony of creatures. An adaptive genetic algorithm is employed in Bel-GAlg with the probability of crossover varying along the solution path. After a generation has been fully populated and the fitness of all creatures in that generation are known, an adaptive probability for crossover P_c for every chromosome belonging to an individual creature is then assigned according to the formula

$$P_c = \frac{1}{P} \begin{cases} P^c_{\max} - \frac{(P^c_{\max} - P^c_{\min})(f - f_{\text{avg}})}{f_{\max} - f_{\text{avg}}} & f \geq f_{\text{avg}}, \\ P^c_{\max} & f < f_{\text{avg}}, \end{cases}$$

where $P^c_{\max}$ and $P^c_{\min}$ are the maximum and minimum probabilities for crossover occurring within a creature, which are exported from the module `BelGAlg.Creatures` through the constants `maxProbabilityOfCrossover` and `minProbabilityOfCrossover`. There they are assigned values of 0.9 and 0.6, respectively. Here P designates the number of chromosomes (or material parameters in a model) that belong to a creature. The fitness of the creature is denoted as f, while f_{avg} and $f_{\max}$ correspond to the average fitness of that generation's population and the fitness of its *elite* creature. This formula is akin to an annealing protocol, a consequence derived by Schmitt [7] to ensure convergence.

At the time of reproduction, the probability of crossover applied to an offspring is the average of the probabilities of crossover belonging to its parents.

6 Genomes

A genome is the whole cache of biological memory. It contains the entire evolutionary history of a living organism. In a GA, it contains the genetic imprint of a creature.

This GA differs from SGA by including the concept of a genome, taken here to be the collection of all chromosomes belonging to an individual. Recall that each model parameter associates with its own chromosome. Therefore, there is a one-to-one correlation between the array of numbers that are a model's parameters and the array of chromosomes that comprise its genome. Different models will have different genome; they are akin to being different species. In contrast, the human has 23 diploid chromosomes. Each is comprised of two gene strings. One string is taken from each parent.

Module `BelGAlg.Genomes` exports the ancillary data type `Genotype`, which is an **array * of** `BelGAlg.Chormosomes.Chromosome` for which the predominant type `Genome` is a wrapper. Type `Genome` implements two definitions; they are: `[]` and `Bel.Entity`.

Via `Genome`'s implementation of the Zonnon indexer `[]`, access is gained to the various chromosomes in a genome, with admissible values belonging to the interval $[0, \texttt{Strands} - 1]$, where method `Strands` returns the number of chromosomes in the genome, i.e., the number of parameters to be estimated.

Method `Clone` returns an exact duplicate of itself. Assignment := returns a shallow copy of a `Genome`. (`Genome` does not implement the `Bel.Object` interface.)

Method `Equals` compares two genome, and is called by the overloaded operators "=" and "#". It tests for equality of gene expression.

Methods `Mutate`, `Encode` and `Decode` all forward their requests down to its chromosomes, where the appropriate actions are taken.

Module `BelGAlg.Genomes` exports two procedures; they are `Crossover` and `Similarity`. `Crossover` redirects the request to its underlying individual chromosomes for action. `Similarity` compares two genome, returning the fraction that are the same, i.e., the number of genes where both genome have the same allele, divided by the number of genes in the genome. This value is reported on for the elite creature of each generation in the final report, cf. App. A. Because clones are not permitted in this GA (they are in some), its value will always be below 1.0. As the solution converges, this quantity ought to become close to 1.

7 Creatures

We now move up to the level of a physical being, an entity, or an organism. The module `BelGAlg.Creatures` exports type `Lineage`, which keeps track of an individual's assets. This type makes available a creature's `birthID`, which is akin to a social security number; its `fitness`, which determines its chances for mating; its parameters, which establish its genetics; the similarity of its parent's genome `parentSimilarity`, which is their fraction of allele with the same gene expression; its `probabilityOfMutation`, which adapts dynamically (i.e., it anneals) with the solution; and its `probabilityOfCrossover`, which also adapts with the evolving solution. `Lineage` implements the `Bel.Object` interface because BelGAlg uses a `Bel.List` to store the lineage of all creatures belonging to a colony so that a report (cf. App. A) can be written.

The main type exported by this module is a `Creature`, which implements the `Bel.Entity` interface. Method `Clone` returns a deep copy of itself; whereas, assignment := returns a shallow copy. Method `Create` should be called immediately after a creature is created via the **new** command. Two arguments are passed with this call. The first, `fixedParameters`, is an array of those model parameters that the user chooses to fix, i.e., the optimizer will not vary these. If all of the parameters are to vary, then `fixedParameters` is **nil**. The second argument, `varyParameters`, is a boolean array designating which variables are to be held fixed and which are allowed to vary, as they are passed to a model of type `BelGAlg.Statistics.Model`. Say, e.g., that your model has 5 parameters, the second and fifth of which you can get reliably from a graphical method. In this case, `varyParameters` would be a boolean array of length 5 with values $\{T, F, T, T, F\}^T$.

Each individual has a unique phenotype that associates with a unique set of model parameters

that are stored in its `lineage` as `parameters`. This array of the model's parameters contains both those that are fixed and those that have been allowed to vary.

Two individuals can be discerned as being clones of one another via the method `Equals`, which is called by the overloaded operators "=" and "#".

An individual's genome can be retrieved with method `Get`.

Probabilities for mutation and crossover, as they pertain to a specific creature, are assigned by the method `AssignProbabilities`, where the first argument specifies the maximum fitness of the population and the second argument specifies the average or mean fitness of the population.

An individual can come into being via three different routes. `Procreate` is used to create the first colony of individuals, and to create immigrants from outside the colony. `Procreate` introduces a Monti Carlo event into the optimization process. It requires arguments that specify the lower and upper bounds defining the interval that each parameter is allowed to vary over, and the number of significant figures in accuracy hoped for in the parameters' final values.

The second way an individual can be created is by an `Alien` migrating into the population. An alien individual has the same arguments that a procreated individual does, plus it specifies what its phenotype is (i.e., its model parameters); whereas, a procreated individual has its phenotype assigned by chance. An alien can be thought of as 'seeding' a population with an individual thought to have good fitness. In a certain sense, BelGAlg seeds each new generation with an alien—the most fit or elite creature from the previous generation.

The third and final way to create an individual is to `Conceive` it through the mating of two distinct individuals, i.e., its parents, which are the first two arguments passed in `Conceive`. This method produces a new `Creature`, viz., their child. It also updates the counters `numberOfMutations` and `numberOfCrossovers`, whose tallies are statistics tabulated in the report (cf. App. A). A call to `Conceive` may also introduce an immigrant, at a probability of once per generation, as an aid to maintaining a diverse genetic pool within a population and to help ensure convergence [9]. Clones are not permitted within a population. In some GAs [5], convergence occurs when the whole population consists of clones of a single creature. That philosophy is not adopted here.

8 Colonies

The highest level in the hierarchy of our GA (there are five levels) is a population of creatures that makes up the colony at a moment in its history. Pairs of creatures from an existing colony are selected for mating through tournament play, whose offspring will comprise the next generation for the colony. The contestants for tournament play are selected at random from the existing population of individuals, with the most-fit contestant from each tournament of play being selected to mate. The number of contestants s in tournament play is set at 2% of the population size, but must be at least 3 in count.

Some GAs allow their population size to vary from one generation to the next. However, the colony size n of this GA is held fixed over all the generations, assigned internally according to the formula [3]

$$n = \lceil \chi^k (k \ln \chi + \ln \ell) \rceil,$$

where χ is the cardinality of the gene alphabet being used (e.g., 2 for haploid, 3 for diploid, etc.), k is the bounding dimension for which all schemata up to that length are to be successfully sampled, recombined, and re-sampled, while ℓ is the number of genes in the genome. The user affects this value through his/her choices for the variables `dimensionOfSchemata` and `numberOfSignificantFigures`, the former of which is assigned to k, and the latter through its influence on the number of genes ℓ required by the encoder/decoder.

A truly remarkable result is that the number of generations needed for convergence t_c can be estimated *a priori* via the formula [3]

$$t_c = \lceil \sqrt{\ell}\ \ln n / \ln s \rceil,$$

where ℓ is the number of genes in the genome, n is the number of creatures that constitutes a population, and s is the number of contestants in tournament play. After t_c generations have been created, a message will be displayed in the terminal window providing the user with certain information to help them determine if more generations are needed, or if the solution is acceptable. Although theory specifies t_c, in practice this predetermined number of generations may not be enough to achieve convergence.

Module `BelGAlg.Colonies` exports two types. The predominant type, `Colony`, places a wrapper around `Inhabitants`, which is an **array * of** `BelGAlg.Creatures.Creature`. `Colony` implements the `Bel.Entity` interface.

To `Create` a `Colony` requires a whole slew of arguments. The first five are the same as those required by `Statistics.Configure`. After that there are: an array (set to **nil** if of zero length) containing parameters whose values the optimizer cannot adjust, a set of alien parameters introduced to seed the optimizer, lower and upper bounds to establish search intervals over which an optimal set of parameters is thought to be found. (The last three of these arrays pertain to only those parameters whose values the optimizer is allowed to adjust.) This is followed by the dimension of schemata that are to be captured, and the number of significant figures in accuracy that are desired in the solution.

Method `Propagate` advances the colony to its next generation.

Method `Parameters` returns the phenotype from the elite individual in the colony for the current generation.

The remaining methods are used to write a report to file. They provide a condensed version for the history of a colony, along with a number of useful statistics, and the optimum parameters found in that search. The number of individuals that lie within intervals of $\pm$ a precision loss of half the number of significant figures, which is documented in the report, provides a sense of how well a particular optimization run performed. As a rule of thumb, at least 10% of the total population size ought to lie within this range. Smaller values suggest that one needs to refine one's search strategy. `ReportHeader` is to be called before the first generation is brought into existence. `ReportBody` can be called after the arrival of any new generation. And `ReportFooter` is to be called after the final generation has been born and reported on. Appendix A contains an example of such a report. Method `CheckOnTheState` reports the current state of the optimizer to a terminal window to allow the user to either terminate or continue with an optimization.

9 Genetic Algorithm Driver

Module `BelGAlg.GeneticAlgorithm` exports two funtions: `Optimize` and `EasyOptimize`. They are the drivers of BelGAlg. They return the optimal set of parameters found by the genetic algorithm. There are twelve arguments in `Optimize` that must be sent to this driver, and six for `EasyOptimize`. All twelve are discussed in some detail, as this is the interface that most programmers will use in order to obtain an optimal set of estimated parameters. `EasyOptimize` calls on a subset of these twelve, using defaults for the other six. If the defaults of `EasyOptimize` are inadequate, then the general `Optimize` procedure needs to be called. In most cases, however, `EasyOptimize` works just fine.

The first argument of both `Optimize` and `EasyOptimize` is a string that contains the name of a file where the report is to be written. It will be placed in the `iofiles` directory beneath the directory where your executable code resides. The file will be given a `.txt` extension.

Their second argument is a matrix that contains the controlled (or independent) variables belonging to the experimental data sets that are being used to fit the model against. These data drive the user-defined model whose parameters are being estimated.

Their third argument is a matrix that contains the output (or dependent) variables belonging to the experimental data sets that are being used to fit the model against. The GA will seek an optimal fit to these data by adjusting the model's parameters.

The fourth argument in `Optimize` is an integer specifying how many data points are to be used by the optimizer. All other data are to be decimated. The chosen data are selected to be roughly equidistant from one to the next along their experimental XY (input/output) curves. If `decimateTo` is less than the number of columns in the response matrix (the third argument) then decimation will occur. This requires that the dimensions of the control and response matrices must be the same. These dimensions can differ only when there is to be no decimation, in which case `decimateTo` is to be set to a value that is greater than or equal to the number of columns in the response matrix.

The fifth argument in `Optimize`, and the fourth in `EasyOptimize`, is a user-defined instance of the procedure type exported as `Model` from module `BelGAlg.Statistics`. Here is where the programmer must provide their model whose parameters are to be estimated. Such procedures need to evaluate the model at all N experimental data points, and return their results. The model itself can be a function, a differential equation, an integral equation, or whatever. But whatever it is, it can only interact with the GA through the `Model` interface.

The sixth argument of `Optimize` provides a boolean array that tells the optimizer how to splice together the fixed and adjustable parameters into a single array of parameters that can then be passed onto the numerical model. If, e.g., the first element of the array were *true*, then the first element in the composite array would come from the first element in the array of varied parameters. If the second element were *false*, then the second element of the composite array would come from the first element in the array of fixed parameters. If the third element were *true*, the the third element of the composite array would come from the second element in the array of varied parameters, and so on.

The seventh argument of `Optimize` lists those model parameters whose values are held fixed, i.e., that the optimizer cannot adjust. This can be an array of length zero, in which case all parameters are adjustable and `fixedParameters` is set to **nil**. All parameters vary whenever `EasyOptimize` is called. The next three arguments pertain only to those parameters that can be adjusted.

The eighth argument of `Optimize` is a set of alien parameters for seeding the GA. These may be your best guess at what you expect they could be, or they may be parameters from a previous run that terminated too soon, for whatever reason, or they may be an optimum set obtained from a prior experimental data set. Whatever the situation may be, this alien becomes the colony's Adam and Eve. `EasyOptimize` takes the median value of the range set for each parameter (via the next two inputs) as the parameters for its alien being.

The ninth argument of `Optimize`, and the fifth argument of `EasyOptimize`, contains the minimum or lower bounds for the set of adjustable parameters. No parameter is allowed below this boundary.

The tenth argument of `Optimize`, and the sixth and final argument of `EasyOptimize`, contains the maximum or upper bounds for the set of adjustable parameters. No parameter is allowed above this boundary. Between these lower and upper bounds lie intervals in which the optimum set of parameters is thought to reside. Typically, one will start with large intervals and then refine them with successive optimizations if the final $\mathcal{R}^2$ statistic is thought to be too small. This honing in on a solution can be a useful strategy whenever there exist multiple minima in a solution space, as is the case in the example that follows.

The eleventh argument of `Optimize` establishes the maximum length of schemata, i.e., the building blocks, that you want the optimizer to be able to handle in its search for a global minimum. Typical values lie between 3 and 8. The default in `EasyOptimize` is set at 6.

The twelfth and final argument of `Optimize` specifies the number of significant figures that you would like the optimizer to converge to. This has a significant impact on the lengths that chromosomes take on, and therefore, on the size of the population, too. Values between 2 and 6 are common, with the default in `EasyOptimize` being set at 4.

9.1 An Example

Appendix A provides an example report for the following test problem, which is included in the test directory of the BEL-GAlg software. This problem was given to my engineering students for their final project in the winter semester of 2011.

The author [10] recently derived a hypo-elastic theory for describing the passive response of the soft tissue in lung, a.k.a. the parenchyma. In that document he solved the boundary value problem (BVP) of simple extension, i.e., the stretching of a rod, as it applies to this new material model. Although this is not an experiment typically used to characterize lung, nor is it representative of natural lung response, nevertheless it is the most prevalent experiment used to characterize almost all materials. The outcome was a Riccati differential equation with constant coefficients whose solution is

$$\sigma = E\frac{\sinh(a\epsilon)}{a\cosh(a\epsilon) - b\sinh(a\epsilon)},$$

with material constants

$$E > 0, \quad 0 < b < a,$$

where σ is the stress (i.e., the output) and ϵ is the strain (viz., the input), while a, b, and E are material parameters with E being the elastic or Young's modulus. Parameters a and b are dimensionless, while E has units of stress (force over area).

The hypothetical data set that was supplied to my students is listed in Table 1, less the initial condition of $(\epsilon_0, \sigma_0) = (0, 0)$ that is trivially satisfied. A typical outcome from such an optimization is given in App. A, wherein parameter 1 associates with a, parameter 2 with b, and parameter 3 with E. It is worth while to point out that no two runs will be identical. GAs are probabilistic in origin. Nevertheless, their final results ought to be very similar to one another.

`EasyOptimize` was used, with the final pass of the optimizer being configured to use the parameter bounds

$$\theta_{min} = \left\{ \begin{array}{c} 0.1 \\ 0.1 \\ 0.1 \end{array} \right\} \quad \theta_{max} = \left\{ \begin{array}{c} 1.5 \\ 1.5 \\ 1.5 \end{array} \right\},$$

ϵ	0.1	0.2	0.3	0.4	0.5
	0.6	0.7	0.8	0.9	1.0
	1.1	1.2	1.3	1.4	1.5
	1.6	1.7	1.8	1.9	2.0
σ	0.050	0.111	0.193	0.290	0.349
	0.450	0.559	0.622	0.744	0.835
	1.032	1.144	1.266	1.396	1.409
	1.494	1.625	1.675	1.700	1.710

Table 1: Hypothetical experimental data for the stretching of lung. Strain ϵ is dimensionless, while stress σ has units of cm H_2O (or gm/cm^2).

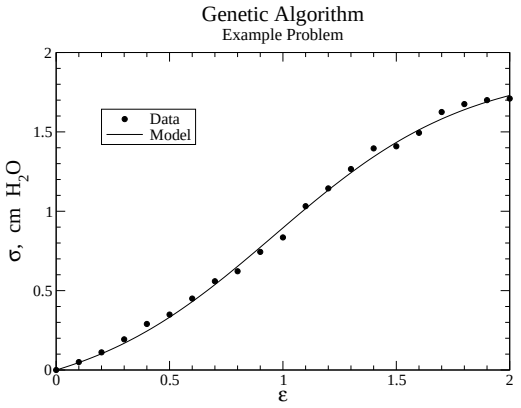


Figure 1: GA fit to raw data, $\mathcal{R}^2 = 0.996$.

which just happened to be the same for all three parameters. The dimensions of this problem are: $C = 1$, $N = 20$, $O = 20$, $P = 3$ and $R = 1$. The acquired fit of the model is displayed in Fig. 1.

The example run of App. A required 19 generations to reach theoretical convergence, given a population size of 511. Values belonging to the most-fit creature in the i^{th} generation are reported; namely: the fitness Φ_i, the fraction of genes with the same allelic values between the elite's parents, and the adjusted $\mathcal{R}_i^2$ statistic. This is followed by a section that displays some of the vital statistics of the run, viz., the numbers of crossovers, mutations, and immigrants, the parameter estimates along with measures of their sensitivity, and the Gray codes held by the elite creature over the generations.

The 'viable population size' of 4754 is the total number of individuals over the whole history of the colony whose parameters all fell within the sensitivity bounds listed. If one takes the population

size, minus one for the elite creature, and multiplies that by the number of generations one arrives at the total number of creatures over the history of the colony, adding 1 for the alien, which in the case of this example was 9691, as reported. The average probability for crossover, for the whole colony, is then $7942/9691 = 0.8195$, which lies between $P^c_{\max} = 0.9$ and $P^c_{\min} = 0.6$, as expected. Likewise, its average probability for mutation is determined to be $8112/(45 \times 9691) = 0.0186$, which lies between $P^m_{\max} = 0.1$ and $P^m_{\min} = 0.001$, again as expected.

The time required to run this optimization problem was on the order of a few seconds.

10 History of Changes

BEL version releases wherein no change took place in BelGAlg are omitted. Version numbering corresponds with the version releases of BEL.

10.1 Changes to Version 4.1

Bug fixes. Procedures `GeneticAlgorithm.Optimize` and `GeneticAlgorithm.EasyOptimize` are now functions. They return the optimal set of parameters found.

10.2 Changes to Version 4.0

The concept of annealing was introduced via adaptive measures for the probabilities of crossover and mutation. The process by which children are conceived was completely reworked, which lead to simpler interface and usage of the optimizer. A capability to continue/stop an optimization run was added to make the overall operation more user friendly. Substantial changes to the interface resulted from fundamental changes introduced through `Bel.Types`.

10.3 Changes to Version 3.3

The theoretical sections deriving the objective function, the $\mathcal{R}^2$ statistic, and the sensitivity analysis have been completely reworked. The statistics are now a tighter representation of what the author perceives as being 'reality'. This major overhaul of module `BelGAlg.Statistics` did somewhat affect its interface, and consequently, the interfaces of modules `Colonies` and `GeneticAlgorithm`, too.

10.4 Changes to Version 3.0

Removed `alienParameters` as an argument in procedure `EasyOptimize`.

10.5 Changes to Version 2.3

A major internal change took place in that maximum likelihood estimates, instead of maximum R^2 values, are now used as the measure of fitness, i.e., the objective function.

Several bugs in `Bel.GA.Statistics` were fixed, including an error in calculating the covariance matrix, and its interface, i.e., `Bel.GA.Statistics.Covariance`, was changed to make the implementation more efficient.

10.6 Changes to Version 2.1

A bug was fixed that resulted from BEL GA being extracted from BEL to become a standalone library.

10.7 Changes to Version 2.0

The genetic algorithm was extracted from the core BEL library as an application library.

References

[1] Gutknecht, J., 2009, *Zonnon Language Report*, ETH Zürich, Switzerland, 4th ed., available online at www.zonnon.ethz.ch.

[2] Goldberg, D. E., 1989, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Boston.

[3] Goldberg, D. E., 2002, *The Design of Innovation: Lessons learned from and for Competent Genetic Algorithms*, vol. 7 of *Genetic Algorithms and Evolutionary Computation*, Kluwer, Boston.

[4] Sivanandam, S. N. and Deepa, S. N., 2008, *Introduction to Genetic Algorithms*, Springer, Berlin.

[5] Schmitt, L. M., 2001, "Theory of genetic algorithms," Theoretical Computer Science, **259**, pp. 1–61.

[6] Doehring, T. C., Carew, E. O., and Vesely, I., 2004, "The effect of strain rate on the viscoelastic response of aortic valve tissues: A direct-fit approach," Annals of Biomedical Engineering, **32**, pp. 223–232.

[7] Schmitt, L. M., 2004, "Theory of genetic algorithms II: models for genetic operators over the string-tensor representation of populations and convergence to global

optima for arbitrary fitness function under scaling," Theoretical Computer Science, **310**, pp. 181–231.

[8] Michalewicz, Z., 1996, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd ed., Springer, Berlin.

[9] Yao, L. and Sethares, W. A., 1994, "Nonlinear parameter estimation via the genetic algorithm," IEEE Transactions on Signal Processing, **42**, pp. 927–935.

[10] Freed, A. D. and Einstein, D. R., 2011, "Viscoelastic model for lung parenchyma for multi-scale modeling of respiratory system, Phase I: Hypo-elastic model for CFD implementation," Technical Report PNNL-20340, Pacific Northwest National Laboratory, Richland, WA.

A Example GA Output

```
-------------------------------------------------------------
An optimization of 1 random variable over 20 events
           : --------------- most-fit citizen ---------------
generation  :    fitness    : parent similarity : adjusted R^2
-------------------------------------------------------------
     1        1.029834E+003      0.000E+000       9.89765E-001
     2        1.966732E+003      4.667E-001       9.85789E-001
     3        2.623127E+003      4.444E-001       9.93475E-001
     5        3.005862E+003      9.333E-001       9.96597E-001
     7        3.437132E+003      8.000E-001       9.96837E-001
    10        3.484098E+003      6.444E-001       9.96733E-001
    12        3.488700E+003      8.667E-001       9.96963E-001
    15        3.492407E+003      9.333E-001       9.96924E-001
    19        3.495432E+003      8.667E-001       9.96914E-001
-------------------------------------------------------------
total number of genes  = 45
size of the population = 511
number of contestants  = 10
number of generations  = 19
number of individuals  = 9691
number of crossovers   = 7942
number of mutations    = 8112
number of immigrants   = 97
-------------------------------------------------------------
Optimum parameters, +/- 50% loss in significant-digit accuracy
1)  1.071E+000 < 1.082E+000 < 1.093E+000
2)  8.419E-001 < 8.504E-001 < 8.589E-001
3)  4.505E-001 < 4.551E-001 < 4.596E-001
Out of 9691 creatures, 4754 laid within these intervals.
-------------------------------------------------------------
gen : par)   -- genotype --
  1 :   1)   010010001101111
        2)   011001000101001
        3)   011111010101010
 ...
 10 :   1)   111010101000101
        2)   110011001001110
        3)   011000001100101
 ...
 19 :   1)   111010100100101
        2)   110011011010110
        3)   011000001001100
-------------------------------------------------------------
```

B Genetic Algorithm

The procedures inherited from the *[]*, *Bel.Entity*, *Bel.Object*, and *Bel.Typeset* definition interfaces have been implemented in the actual code, but are not repeated here to help keep the exported interfaces concise.

```
module BelGAlg.Statistics;
   import
      Bel.Types as T;
   var {immutable}
      dimC, dimN, dimP, dimR : integer;
   type
      Model = procedure (T.RealVector; T.RealMatrix) : T.RealMatrix;
   procedure FlipHeads (probabilityOfHeads : real) : boolean;
   procedure RandomIntegerBetween (lowValue, highValue : integer) : integer;
   procedure Configure (expInput, expOutput : T.RealMatrix;
                        decimateTo : integer; numericalModel : Model;
                        varyParameters : T.BooleanVector);
   procedure DataFitAgainst (var input, output : T.RealMatrix);
   procedure Theory (parameters : T.RealVector) : T.RealMatrix;
   procedure Fitness (parameters : T.RealVector) : real;
   procedure RSquared (parameters : T.RealVector) : real;
end Statistics.

module BelGAlg.Genes;
   import
      Bel.Entity as Entity,
      Bel.Typeset as Typeset;
   type Allele = integer{8};
   type {value} Gene = object implements Entity, Typeset
      procedure Clone () : Gene;
      procedure Pop () : Allele;
      procedure Put (a : Allele);
      procedure Mutate (probabilityOfMutation : real;
                        var numberOfMutations : integer);
      procedure Equals (g : Gene) : boolean;
   end Gene;
   operator ":=" (var l : Gene; r : Gene);
   operator "=" (l, r : Gene) : boolean;
   operator "#" (l, r : Gene) : boolean;
end Genes.
```

```
module BelGAlg.Chromosomes;
   import
      Bel.Entity as Entity,
      Bel.Typeset as Typeset,
      BelGAlg.Genes as G;
   type Haploid = array * of G.Gene;
   type {ref} Chromosome = object implements [], Entity, Typeset
      procedure Clone () : Chromosome;
      procedure Create (minParameter, maxParameter : real;
                        numberOfSignificantFigures : integer);
      procedure Length () : integer;
      procedure Mutate (mutationProbability : real;
                        var numberOfMutations : integer);
      procedure Decode () : real;
      procedure Encode (phenotype : real);
      procedure Equals (c : Chromosome) : boolean;
   end Chromosome;
   operator "=" (l, r : Chromosome) : boolean;
   operator "#" (l, r : Chromosome) : boolean;
   procedure Crossover (probabilityOfCrossover : real;
                        parentA, parentB : Chromosome;
                        var numberOfCrossovers : integer;
                        var childA, childB : Chromosome);
end Chromosomes.


module BelGAlg.Genomes;
   import
      Bel.Entity as Entity,
      Bel.Types as T,
      BelGAlg.Chromosomes as C;
   type GenoType = array * of C.Chromosome;
   type {ref} Genome = object implements [], Entity
      procedure Clone () : Genome;
      procedure Create (minParameters, maxParameters : T.RealVector;
                        numberOfSignificantFigures : integer);
      procedure Equals (g : Genome) : boolean;
      procedure Strands () : integer;
      procedure Mutate (probabilityOfMutation : real;
                        var numberOfMutations : integer);
```

```
      procedure Encode (phenotypes : T.RealVector);
      procedure Decode () : T.RealVector;
   end Genome;
   operator "=" (l, r : Genome) : boolean;
   operator "#" (l, r : Genome) : boolean;
   procedure Crossover (probabilityOfCrossover : real;
                        parentA, parentB : Genome;
                        var numberOfCrossovers : integer;
                        var childA, childB : Genome);
   procedure Similarity (genomeA, genomeB : Genome) : real;
end Genomes.


module BelGAlg.Creatures;
   import
      Bel.Entity as Entity,
      Bel.Object as Object,
      Bel.Types as T,
      BelGAlg.Genomes as G;
   const
      minProbabilityOfMutation = 0.001;
      maxProbabilityOfMutation = 0.1;
      minProbabilityOfCrossover = 0.6;
      maxProbabilityOfCrossover = 0.9;
   var
      birthNumber : integer;
   type {ref} Lineage = object implements Object
      var
         birthID : integer;
         fitness : real;
         parameters : T.RealVector;
         parentSimilarity : real;
         probabilityOfMutation : real;
         probabilityOfCrossover : real;
   end Lineage;
   type {ref} Creature = object implements Entity
      var {immutable}
         lineage : Lineage;
      procedure Clone () : Creature;
```

```
      procedure Create (fixedParameters : T.RealVector;
                        varyParameters : T.BooleanVector);
      procedure Get () : G.Genome;
      procedure Alien (parameters, minParameters, maxParameters : T.RealVector;
                        numberOfSignificantFigures : integer);
      procedure Procreate (minParameters, maxParameters : T.RealVector;
                        numberOfSignificantFigures : integer;)
      procedure Conceive (parentA, parentB : Creature;
                        var numberOfMutations, numberOfCrossovers : integer);
      procedure Equals (c : Creature) : boolean;
      procedure AssignProbabilities (maxFitness, avgFitness : real);
   end Creature;
   operator "=" (l, r : Creature) : boolean;
   operator "#" (l, r : Creature) : boolean;
end Creatures.

module BelGAlg.Colonies;
   import
      System.IO.StreamWriter as StreamWriter,
      Bel.Entity as Entity,
      Bel.Types as T,
      BelGAlg.Statistics as S,
      BelGAlg.Creatures as C;
   type Inhabitants = array * of C.Creatures;
   type {ref} Colony = object implements Entity
      procedure Create (expInput, expOutput : T.RealMatrix;
                        decimateTo : integer; numericalModel : S.Model;
                        varyParameters : T.BooleanVector;
                        fixedParameters, alienParameters,
                        minParameters, maxParameters : T.RealVector;
                        dimensionOfSchemata,
                        numberOfSignificantFigures : integer);
      procedure Propagate (var converged : boolean);
      procedure Parameters () : T.RealVector;
      procedure ReportHeader (sw : StreamWriter);
      procedure ReportBody (sw : StreamWriter);
      procedure ReportFooter (sw : StreamWriter);
      procedure CheckOnTheState (var finished : boolean);
   end Colony;
end Colonies.
```

```
module BelGAlg.GeneticAlgorithm;
   import
      Bel.Types as T,
      BelGAlg.Statistics as S;
   procedure Optimize (fileNameForReport : string;
                       expInput, expOutput : T.RealMatrix;
                       decimateTo : integer; numericalModel: S.Model;
                       varyParameters : T.BooleanVector;
                       fixedParameters, alienParameters,
                       minParameters, maxParameters : T.RealVector;
                       dimensionOfSchemata,
                       numberOfSignificantFigures : integer)
                       : T.RealVector;
   procedure EasyOptimize (fileNameForReport : string;
                       expInput, expOutput : T.RealMatrix;
                       numericalModel : S.Model;
                       minParameters, maxParameters : T.RealVector)
                       : T.RealVector;
end GeneticAlgorithm.
```