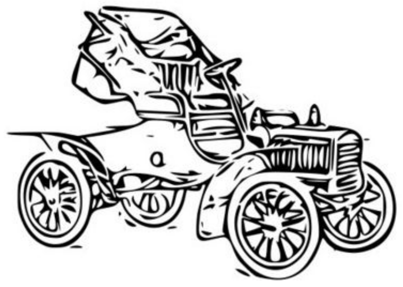
vgGA2.0 User Manual

BODI Report No. 2017-001

Manuel Valenzuela-Rendón

Tecnológico de Monterrey
Monterrey, N.L., Mexico

January 3, 2017



A model C is not a C-model.

Abstract

This reports is the vgGA2.0 User Manual. the virtual gene Genetic Algorithm (vgGA) is a genetic algorithm in which traditional genetic operators are implemented as arithmetic operations over the phenotype. Furthermore, vgGA generalizes the concept of digit by allowing crossover points to fall in values that do not correspond to integer powers of the base being used. vgGA2.0 is the Octave/MATLAB implementation of these ideas. vgGA2.0 is a major overhaul of vgGA1.2 which was only released locally at the Tecnológico de Monterrey. In vgGA2.0 includes many corrections, among others the use of traditional, generalized, and continuous digits, gamma correction for mutation, and crossover with exponential and uniform distributions over values. This release of the vgGA includes a tutorial and several scripts that demonstrate its use.

Contents

Noli, obsecro, istum disturbare.
Archimedes

A note from the author

The vgGA idea was born out of the need to implement a traditional genetic algorithm in Octave and/or MATLAB. The software described in this report was initially developed for research purposes. Later, I used it to teach my courses on Evolutionary Computation. More than striving for efficiency, the code was developed trying to make it clear and easy to modify. The vgGA is OOP, so it is fairly easy to add methods. I strongly recommend that if you need additional functionality of an existing method, you give a new name to the modified method, if possible and not inconvenient.

Since this code has been developed for research and teaching purposes and it is in continuous revision and change, many bugs may remain, and many efficiency improvements are possible. I take no responsibility for any malfunction; bugs, errors, and suggestions may be communicated to me at `manuel.valenzuela.r@gmail.com`. Commercial use of this software is strictly prohibited without my permission. For academic purposes, it can be used and modified as desired, nevertheless, appropriate acknowledgment will be greatly appreciated.

If you would like to skip all the theoretical stuff, you can jump directly to section 8 for a quick tutorial on the vgGA use, and then come back here.

1 Introduction

This report explains the use and implementation in Octave/MALTAB of vgGA2.0. The virtual gene Genetic Algorithm (vgGA) (Valenzuela-Rendón, 2003) is based on the idea that traditional genetic operators of crossover and mutation, which usually are implemented over the genotype, can be mapped to arithmetic operations over the phenotype, where the phenotype is a vector of integers or reals. Thanks to this mapping, it is possible to implement chromosomes of any integer base, not only binary, without actually implementing chromosomes as strings of characters. Furthermore, vgGA extends the idea of traditional digits to *generalized* digits and to *continuous* digits. In this way, the vgGA is a superset of traditional genetic algorithms.

vgGA2.0 is a major overhaul of vgGA1.2 which was locally released to students at the Tecnológico de Monterrey. As explained below, the mutation clock has been modified to correctly implement traditional, generalized, and continuous digits with exponential and uniform distributions over values, using the gamma correction. Also, normalized crossover with correction has been implemented.

2 Review of the vgGA idea

In a traditional genetic algorithm (Goldberg, 1989), individuals are strings in a low cardinality alphabet, usually binary; new individuals are created through one-point crossover and mutation. For parametric optimization, real numbers can be represented in a binary chromosome by a linear mapping. For a variable in the domain $[x_{\min}, x_{\max})$, the binary N-digit chromosome of base B is transformed to an integer x_{integer} and transformed to the real number represented by the following formula[1]:

$$x_{\text{real}} = x_{\text{integer}} \left(\frac{x_{\max} - x_{\min}}{B^{\ell}} + x_{\min} \right) \tag{1}$$

One-point crossover exchanges segments of the parents chromosomes, and mutation complements individual bits. In the vgGA, these two genetic operators are mapped to arithmetic operations over the phenotype of the individuals, i.e., over the integer numbers or real numbers the chromosomes represent. As an example of crossover, consider the following two parents:

$$p_1 = 1443_5 = 248 \tag{2}$$
$$p_2 = 4310_5 = 580 \tag{3}$$

A crossover after digit 2 (digits are number right to left) can be obtained in the following way:

$$\chi(p_1, p_2, n) = p_1 \bmod n - p_2 \bmod n = 248 \bmod 5^2 - 580 \bmod 5^2 = 23 - 5 = 18 \tag{4}$$

$$\begin{aligned} h_1 &= \text{crossover}(p_1, p_2, n) = p_2 + \chi(p_1, p_2, n) && (5)\\ &= 580 + 18 = 598 = 4343_5 && (6)\\ h_2 &= \text{crossover}(p_2, p_1, n) = p_1 - \chi(p_1, p_2, n) && (7)\\ &= 248 - 18 = 230 = 1410_5 && (8) \end{aligned}$$

where h_1 and h_2 are the offspring produced by this crossover. As an example of the crossover of reals, consider the following real individuals in $[-1, 1)$ represented with binary 4-bit chromosomes:

$$r_1 = -0.875\ (0001_2) \tag{9}$$
$$r_2 = 0.5\ (1100_2) \tag{10}$$

The crossover of these individuals after bit 2 which corresponds to the value of $k = -0.5$ is obtained in the following way:

$$\begin{aligned} \chi(r_1, r_2, k, r_{\min}) &= (r_1 - r_{\min}) \bmod (k - r_{\min}) - (r_2 - r_{\min}) \bmod (k - r_{\min}) && (11)\\ &= (-0.875 + 1) \bmod (-0.5 + 1) - (0.5 + 1) \bmod (-0.5 + 1) && (12)\\ &= 0.125 \bmod 0.5 - 1.5 \bmod 0.5 = 0.125 && (13) \end{aligned}$$

$$\begin{aligned} h_1 &= \text{crossover}(r_1, r_2, k, r_{\min}) = r_2 + \chi(r_1, r_2, k, r_{\min}) && (14)\\ &= 0.5 + 0.125 = 0.625\ (1101_2) && (15)\\ h_2 &= \text{crossover}(r_2, r_1, k, r_{\min}) = r_1 - \chi(r_1, r_2, k, r_{\min}) && (16)\\ &= -0.875 - 0.125 = -1\ (0000_2) && (17) \end{aligned}$$

[1]The linear mapping for real parameters described by Goldberg (1989, chapter 3) is defined by the following formula:

$$x_{\text{real}} = x_{\text{integer}} \left(\frac{x_{\max} - x_{\min}}{B^N - 1} + x_{\min} \right)$$

With this mapping, the integers $\{0, 1, \ldots B^N\}$ is mapped to the reals $[x_{\min}, x_{\max}]$, therefore, the maximal chromosome represents the real value $x_{\min}$, while in the vgGA this value cannot be represented.

In vgGA mutation, a segment of digits is removed and replaced by a randomly chosen value which could be the one that was there before mutation. As an example of mutation, consider the individual

$$p = 2142_5 = 297 \tag{18}$$

Mutation of the segment of 2 digits starting at digit 2, i.e. digits 2 and 3, can be obtained in the following way:

$$\begin{aligned} \text{mutation}(p, n, \delta) &= L(p, n) + H(p, n\delta) + \{0, 1, \ldots, \delta - 1\}_{\mathcal{R}} && (19)\\ &= \text{low}(297, 5^1) + \text{high}(297, 5 \cdot 5^2) + \{0, 5, 10, 15, 20, 25, 30 \ldots, 120\}_{\mathcal{R}} && (20)\\ &= 0002_5 + 2000_5 + \{0000_5, 0010_5, 0020_5, 0030_5, 0040_5, 0100_5, 0110_5 \ldots, 0440_5\}_{\mathcal{R}} && (21) \end{aligned}$$

where

$$\begin{aligned} \text{low}(297, 5^1) &= 297 \bmod 5^1 = 2 = 0002_5 && (22)\\ \text{high}(297, 5^3) &= 297 - 297 \bmod 5^3 = 250 = 2000_5 && (23) \end{aligned}$$

and $\{\cdot\}_{\mathcal{R}}$ indicates that one of these values will be chosen randomly.

The vgGA generalizes the traditional genetic algorithm in the following ways:

- **Exponential and uniform distribution of crossover points**
  In a traditional genetic algorithm, crossover points are chosen randomly, were all crossover sites are equally probable. The *values* (k in formulas 11, 14, and 16) are therefore *distributed exponentially over the value.* vgGA can also distribute crossover points uniformly over the value.

- **Traditional, generalized, and continuous digits**
  In a traditional genetic algorithm, crossover and mutation occur at values that are integer powers of the base in which the chromosome is represented. In the vgGA, we call this using *traditional digits.* The arithmetic formulas for crossover and mutation on integers can take any integer values; we call this using *generalized digits.* Furthermore, and for reals, crossover and mutation can fall at any value; we call this using *continuous digits.*

- **Segments**
  In the vgGA, individuals are stored as their phenotype, as opposed to traditional genetic algorithms where the genotype (or chromosome) is usually stored and operated on. Individuals are vectors of integers or reals. Each component of the phenotype will be called a *segment*. The maximum integer value of a segment is equivalent to 2^{52} in whatever base the segment is expressed, and this determines the maximum number of digits in a segment.

  When performing crossover, a segment is chosen with a probability proportional to the number of digits in that segment, and then a crossover value is chosen with the probability distribution desired. For continuous digits, a segment is randomly chosen. Crossover can fall between segments.

- **Mutation clock**
  In a traditional genetic algorithm, the mutation clock as suggested by Goldberg (1989) is the idea that instead of deciding if each individual bit is to be mutated, once a bit is mutated, the

```
Ifuns/                  Gfuns/
   icrossAt.m              gdigit.m
   idigit.m                ghigh.m
   ihigh.m                 ghighh.m
   ihighh.m                glow.m
   ilow.m                  gMutate.m
   iMutate.m               gmutValues.m
   imutValues.m            gsegment.m
   isegment.m
```

Figure 1: Listing of the `Ifun/` and `Gfuns/` directories.

next bit to be mutated is determined using the appropriate probability distribution. Using the geometric distribution, the next mutation occurs after

$$\left\lceil \frac{\ln\left(1 - \mathrm{rand}()\right)}{\ln(1 - p_m)} \right\rceil \tag{24}$$

bits, where p_m is the mutation probability per bit, and $\mathrm{rand}()$ returns a random number in $[0, 1)$ with uniform distribution. The vgGA takes this idea and implements it over the value for traditional and generalized digits. vgGA2.0 defines mutation in terms of the expected number of mutations per segment, or m_s, which the user defines for each segment.

This section is only a brief review of the vgGA. For further information, the reader is referred to (Valenzuela-Rendón, 2003).

3 `Ifuns/` and `Gfuns/` directories

We will assume that the vgGA is installed in the root directory `vgGA2.0/`. The `Ifuns/` and `Gfuns/` directories contain functions that operate over integer phenotypes for traditional and generalized digits. Functions in `Ifuns/` implement traditional digits, and functions in `Gfuns/` implement generalized digits. These functions serve only to show how the ideas that gave birth to the vgGA where generalized, and these functions are not needed for the usual operation of the vgGA; thus their use will not be explained in this document. Figure 1 shows the listing of `Ifuns/` and `Gfuns/`. The equivalent functions in `vgGA2.0/` can work with integers and reals for traditional, generalized, and continuous digits.

4 Basic functions

The directory `vgGA2.0/` contains basic functions needed for the operation of the vgGA, and the idea of implementing crossover and mutation over the phenotype. Most of these functions operate over real phenotypes (y therefore, also over integer phenotypes) for traditional and generalized digits. Through this report, default values will be indicated by an equal sign, for example, `rMin=0` indicates that if not given, the parameter `rMin` will take the value of `0`.

`logB(x,B=2)` Returns $\log_{\mathtt{B}} \mathtt{x}$.

`low(r,k,rMin=0)` Returns the *lower* part of a number `r` up to the value `k`.

`high(r,k,rMin=0)` Returns the *higher* part of a number `r` strating from value `k`.

`highh(r,k,rMin=0)` Returns the value of the higher part of a number `r` starting at value `k`.

`digit(p,n,B)` Returns the digit of weight `n`/`B` of base `B` of the integer `p`.

`segment(r,k,delta=2,rMin=0)` Returns the segment of width `delta` starting at, and including, the value `k` of the number `r`. `delta` must take an integer value greater than 1.

The functions low and high are used to implement crossover and mutation. As an example, consider the integer individual $586 = 4321_5$:

$$\texttt{low}(586, 5^2) = 11 = 0021_5 \tag{25}$$
$$\texttt{high}(586, 5^2) = 575 = 4300_5 \tag{26}$$
$$\texttt{digit}(586, 5^2, 5) = 2 = 2_5 \tag{27}$$
$$\texttt{segment}(586, 5^1, 5^2) = 17 = 32_5 \tag{28}$$

4.1 Crossover

The following functions are used to implement crossover:

`crossPoint(type,digits,dist,N,B,rMin,rMax,D=0)` Obtains a random cross site value according to the type of digits being used (traditional, generalized or continuous) and distribution (exponential or uniform) . Flag `D` should be set to `1` if crossing last segment and the point between the last digit and first digit should not be considered as a possible cross site.

`[h1 h2]=crossAt(r1,r2,k,rMin=0)` Performs a crossover of the numbers `r1` and `r2` with a minimal of `rMin` at the value `k`, y returns offspring `h1` and `h2`. Crossover is defined by the following formulas:

$$h_1 = \text{crossover}(r_1, r_2, k) = r_2 + \chi(r_1, r_2, k, r_{\text{min}})$$
$$h_2 = \text{crossover}(r_2, r_1, k) = r_1 - \chi(r_1, r_2, k, r_{\text{min}})$$

where

$$\chi(r_1, r_2, k, r_{\text{min}}) = (r_1 - r_{\text{min}}) \bmod (k - r_{\text{min}}) - (r_2 - r_{\text{min}}) \bmod (k - r_{\text{min}})$$

`[h1 h2]=crossAtnorm(r1,r2,k,rMin=0)` Performs a normalized crossover of the numbers `r1` and `r2` with a minimal of `rMin` at the value `k`, y returns offspring `h1` and `h2`. Normalization is necessary to reduce roundoff errors when performing crossover of real individuals.

Normalized crossover is defined by the following formulas:

$$h_1 = \text{crossover}_{\text{norm}}(r_1, r_2, k, \Delta\bar{r}, r_{\text{min}}) = r_2 + \bar{\chi}(r_1, r_2, k, \Delta\bar{r}, r_{\text{min}})\Delta\bar{r}$$
$$h_2 = \text{crossover}_{\text{norm}}(r_2, r_1, k, \Delta\bar{r}, r_{\text{min}}) = r_1 - \bar{\chi}(r_1, r_2, k, \Delta\bar{r}, r_{\text{min}})\Delta\bar{r}$$

where

$$\bar{\chi}(r_1, r_2, k, \Delta\bar{r}, r_{\text{min}}) = \left(\frac{r_1}{\Delta\bar{r}} - \frac{r_{\text{min}}}{\Delta\bar{r}}\right) \bmod \left(\frac{k}{\Delta\bar{r}} - \frac{r_{\text{min}}}{\Delta\bar{r}}\right) - \left(\frac{r_2}{\Delta\bar{r}} - \frac{r_{\text{min}}}{\Delta\bar{r}}\right) \bmod \left(\frac{k}{\Delta\bar{r}} - \frac{r_{\text{min}}}{\Delta\bar{r}}\right)$$
$$\Delta\bar{r} = \frac{r_{\text{max}} - r_{\text{min}}}{B^N}$$

Normalized crossover can only be performed on traditional and generalized digits. Normalized crossover is not defined in (Valenzuela-Rendón, 2003).

`[h1 h2]=crossAtc(r1,r2,k,rMin,rMax,B,Dr,correction=4)` Performs a crossover with correction of the numbers `r1` and `r2` at the value `k`, y returns offspring `h1` and `h2`. This function calls `crossAt` and `crossAtnorm`.

A correction for crossover is necessary because crossover for generalized digits may produce invalid individuals. For example, crossover of 15 and 7 at the value 3 produces an offspring of 16 which is invalid for $N = 4$. Crossover of 0.9 and 0.4 at 0.7 produces a child of 1.1 which is invalid in the domain $[0, 1)$. When an invalid offspring is detected, it can be corrected in several different ways. The following are the four corrections implemented in this function:

1. Do not perform crossover.
2. Crossover at the nearest traditional digit.

$$n_c = B^{\lfloor \log_B n \rfloor}$$
$$k_c = B^{\lfloor \log_B ((k - r_{\min})/\Delta\bar{r}) \rfloor} \Delta\bar{r} + r_{\min}$$

   where n_c is the crossover site for integers, and k_c for reals.
3. Make the highest valued child equal to the maximum possible value:

$$h_2 = B^N - 1$$
$$h_2 = r_{\max} - \Delta\bar{r}$$

   for integers and reals, respectively.
4. Make the sum of the offspring equal to the maximum possible value:

$$h_2 = B^N - 1 - h_1$$
$$h_2 = r_{\max} + r_{\min} - \Delta\bar{r} - h_1$$

   for integers and reals, respectively.

`[h1 h2]=crossAtUniform(p1,p2,template,B=2)` Performs uniform crossover (Spears & De Jong, 1991) of the integers `p1` and `p2` of base `B` according to the binary `template`. Uniform crossover can only be take place between integer individual using traditional digits.

4.2 Mutation

The following functions are used to implement mutation:

`mutate(r,k,delta,rMin=0)` Mutates number `r` with a minimal of `rMin` at the value `k` which must be a traditional digit.

The mutation of a segment of width δ starting at, and including value k is defined as:

$$\text{mutation}(r, k, \delta, r_{\min}) = L(r, k, r_{\min}) + H(r, (k - r_{\min})\delta + r_{\min}, r_{\min}) + (k - r_{\min})\lfloor \delta \, \text{rand}() \rfloor.$$

`mutValues(r,k,delta,rMin=0)` Returns all the possible results of applying the `mutate` function. For example, consider the binary individual $1010_2 = 10$. A mutation of width $\delta = 4$ at the value $k = 2$ is equivalent to mutating bits 2 and 3 which can produce any of these results $[1000_2, 1010_2, 1100_2, 1110_2]$:

$$\texttt{mutValues}(10, 2, 4) = [8, 10, 12, 14] \quad (29)$$

`mutateGamma` Returns the mutation of `r` at value `k`. Implements the gamma correction. It can be called in two different ways. For generalized digits,

```
mutateGamma(r,k,N,B=2,delta=B,rMax=B^N,rMin=0)
```

and for continuous digits,

```
mutateGamma(r,k,Inf,delta,rMax,rMin=0)
```

The mutation of generalized digits can produce invalid results. For example, the mutation of 15 at 7 with $\delta = 2$ can produce a result of 22, which for $N = 4$ and $B = 2$ is an invalid individual.

We will call γ the maximum value that $\delta \text{rand}()$ can take so mutation produces a value less than B^N; the following equation expresses the relation between integer p and its γ:

$$L(p,n) + H(p,\delta n) + n\gamma < B^N.$$

therefore,

$$\gamma = \max\left(1, \frac{B^N - 1 - L(p,n) - H(p,n\delta)}{n}\right)$$

We define mutation with *gamma correction* for integers as

$$\text{mutation}(p,n,\delta) = L(p,n) + H(p,n\delta) + n\,B^{\text{randi}(0,\lfloor \log_B \gamma \rfloor)}$$

For real individuals:

$$L(r,k,r_{\text{min}}) + H(r,(k-r_{\text{min}})\delta + r_{\text{min}}, r_{\text{min}}) + (k-r_{\text{min}})\gamma < r_{\text{max}}$$

from which

$$\gamma = \max\left(1, \frac{r_{\text{max}} - \Delta\bar{r} - L(r,k,r_{\text{min}}) - H(r,(k-r_{\text{min}})\delta + r_{\text{min}}, r_{\text{min}})}{k - r_{\text{min}}}\right)$$

We define mutation with *gamma correction* for reals as

$$\text{mutation}(k,r,\delta,r_{\text{min}}) = L(r,k,r_{\text{min}}) + H(r,(k-r_{\text{min}})\delta + r_{\text{min}}, r_{\text{min}}) + (k-r_{\text{min}})\,B^{\text{randi}(0,\lfloor \log_B \gamma \rfloor)}.$$

where $\text{randi}(0,m)$ returns an integer in $[0,m]$.

`mutValuesGamma(r,k,N,B,delta,rMax,rMin=0)` Returns a vector of all the possible results of the mutation of `r` at value `k` considering the gamma correction. The number `r` takes a minimal value of `rMin`. For example, consider the binary individual 1100_2. A mutation of width δ_4 at the value $k = 3$ can produce the values $[1100_2, 1111_2]$:

$$\texttt{mutValuesGamma}(12,3,4,2,4) = [12,15] \tag{30}$$

Compare this with mutation without gamma correction:

$$\texttt{mutValues}(12,3,4) = [12,15,18,21] \tag{31}$$

5 Structure of `population`

vgGA is object oriented programmed. The directory `@population/` defines the class `population` which implements a population of the vgGA. This directory contains all the methods of this class.

The structure of `population` has the following fields:

- `evals`
- `params`
- `best`
- `individual`
- `mutclock`
- `trace`

The field `evals` stores the number of objective function evaluations that have been performed su far; the other fields are described in the following subsections.

5.1 `params`

The field `params` stores parameters that affect the operation of the vgGA. These parameters are explained below:

`max` This flag determines if maximization (`1`) or minimization (`0`) will be performed.

`type` Chromosome type. Can be `'integer'` or `'real'`.

`m` Number of segments in each chromosome.

`N[m]` Vector of the number of digits per segment. The maximum number of digits in a segment is 52 bits or its equivalent in the base being used.

`rMin[m]` Vector of the minimals of the segments.

`rMax[m]` Vector of the maximals of the segments.

`DeltaR[m]` $(r_{\max} - r_{\min})/B^N$.

`ms[m]` Expected number of mutations per segment.

`pm[m]` Vector of the probabilities of mutation per segment.

`pc` Crossover probability.

`B[m]` Vector of the bases of the segments.

`delta[m]` Vector of the mutation widths per segment.

`dist` Crossover points distribution type. Can take the values of `'exponential'` or `'uniform'`.

`digits` Type of digits; can be `'traditional'`, `'generalized'`, or `'continuous'`.

Additionally, the following flags control how a population is displayed:

`dTitle`	Display title?
`dIndiv`	Display individuals?
`dTrace`	Display tracing information?
`dMut`	Display mutation clock?
`dStats`	Display population statistics?
`dBest`	Display best individual found so far?
`dChroms`	Display chromosomes?
`dPhenos`	Display phenotypes?
`dParams`	Display population parameters?

5.2 `best`

The field `best` keeps information of the best individual found so far. It has the following fields:

`r[m]`	Phenotype of the best individual. This a vector of reals (or possibly integers) of length `m`.
`fitness`	Fitness of the best individual found.
`evals`	Number of function evaluations when the best individual was found.

5.3 `individual`

The field `individual` is an array that contains all the individuals in the population. Each element of this array have the following fields:

`r[m]`	Phenotype of the individual. This is a vector of reals (or possibly integers) of size `m`.
`fitness`	Fitness of the individual.

5.4 `mutclock`

The field `mutclock` holds the necessary data to implement the mutation clock of the vgGA.

`nPlus[m]`	value where next mutation will occur (integers)
`kPlus[m]`	value where next mutation will occur (reals)
`mPlus[m]`	digit where next mutation will occur
`DeltaI[m]`	individuals to next mutation
`mMax[m]`	maximum digit for mutation

5.5 `trace`

The field `trace` stores data useful to trace and monitor the operation of the vgGA. t has the following fields:

`nMuts` Number of mutations that have been performed.

`nCross` Number of crossovers that have been performed.

`nISC` Number of inter-segment crossovers. These are crossovers where the crossover point falls between two segments.

6 Methods of `population`

The directory `@population/` contains the methods of `population`.

6.1 Basic methods

The basic methods of the `population` are the following:

`population` This method constructs an empty population with the parameters that specify how the vgGA will run. It can be called in two different manners, depending if building a population of integers

`population('i(nteger)',N,ms,pc,B,delta,dist,digits)`,

or reals

`population('r(eal)', R,N,ms,pc,B,delta,dist,digits)`,

where `R` is a `m` $\times$ 2 matrix that hold the minimal and maximal values for each segment and `N` is a vector of the number of digits per segment. All the other parameters have been discussed in section 5.1, are optional, and take the following default values: `ms` $= 0$, `pc` $= 1.0$, `B` $= 2$, `delta` $=$ `B`, `dist` $=$ `'exponential'`, `digits` $=$ `'traditional'`.

`display` This method displays an instance of `population`.

`setDisplay(pop,displayList)` Modifies the parameters that determine the way `display` behaves. The following can be modified by `setDisplay`:

`'title'` Display title. Default is yes.

`'indiv'` Display individuals. Default is yes.

`'trace'` Display trace information. Default is no.

`'mclock'` Display mutation clock information. Default is no.

`'stats'` Display fitness statistics (average, median, min, max, standard deviation, skewness). Default is yes.

`'best'` Display best individual found so far. Default is yes.

`'chroms'` Display chromosomes. Default is yes. Chromosomes are not displayed for `N=Inf`.

`'phenos'` Display phenotypes. Default is yes.

`'params'` Display population parameters. Default is yes.

For example

```
pop=setDisplay(pop,'best',1,'chroms',0,'params',0)
```

sets the population `pop` to display the best individual, and not display chromosomes and population parameters.

`get` Returns the content of a `population`. Currently implemented fields are `'type'`, `'N'`, `'pc'`, `'pm'`, `'max'`, `'delta'`, `'dist'`, `'digits'`, `'B'`, `'individual'`, `'best'`, `'evals'`, `'r'`, `'fitness'`, and `'rMax'`.

`set` Modifies a field of a `population`. Currently implemented fields are `'type'`, `'N'`, `'pc'`, `'pm'`, `'delta'`, `'digits'`, `'B'`, `'r'` (give individual index), `'max'`, and `'best'`.

`random(pop,n,add='N')` Creates `n` random individuals. If `add` is `'Y'`, these individuals are appended to the population `pop`.

`init(pop)` Initializes a population erasing all the individuals and erasing all information regarding best individual found so far.

`fill(pop,value=0)` Fills the field `r` with a `value` that defaults to 0.

`max(pop)` Loads the value of `1` to the flag of `max` of the field `params` so that maximization is performed.

`min(pop)` Loads the value of `0` to the flag of `max` of the field `params` so that manimization is performed. If minimization with proportional selection (`roulette` or `sus`) is desired, `scale` must be called before selection for minimization to actually occur.

6.2 Crossover and mutation

`crossover(pop,lastTwo='N')` Applies one-point crossover to population `pop`; couples are randomly chosen. If `lastTwo` is `'Y'`, crossover is performed only on the last two individuals in the population (used to implement a steady-state genetic algorithm).

`uniformCrossover(pop,p0,lastTwo=0)` Applies uniform crossover to population `pop`; couples are randomly chosen. Each crossover is performed using a randomly chosen binary template where the probability of zero is `p0`. If `lastTwo` is `'Y'`, crossover is performed only on the last two individuals in the population (used to implement a steady-state genetic algorithm). This method only works for integers and traditional digits.

`mutation(pop,last='N')` Applies mutation to population `pop`. If `last` is `'Y'`, only the last individual in the population is mutated (used to implement a steady-state genetic algorithm).

6.3 Selection

`evaluate(pop,objective_function,start=1,setBestFlag=1)` Evaluates population `pop` using the `objective_function` which receives a vector of the same length than the field `r` of the individuals in `pop`. The population is evaluating starting with the individual in position `start`. If `setBestFlag` is true, the best found is updated.

`evaluateLast(pop,objective_function,N=1,setBestFlag=1)` Evaluates the last `N` individuals in the population using `objective_function` which receives a vector of the same length than the field `r` of the individuals in `pop`. If `setBestFlag` is true, the best found is updated.

`eraseWeak(pop,N=1)` Erases the `N` worst individuals in the population.

`roulette(pop,normalized=0)` Performs roulette wheel selection (Goldberg, 1989) over the population `pop`.

`sus(pop,normalized=0)` Performs stochastic universal sampling (SUS) (Baker, 1987) over the population `pop`.

`linRanking(cMult=2)` Performs linear ranking selection. Constant `cMult` takes the value of 2.0 by default.

`tournament(pop,m=2)` Performs tournament selection (Goldberg & Deb, 1991) of size `m` over the population `pop`. `m` is optional and takes the value of `2` by default.

`btournament(pop,compare_function)` Performs binary tournament selection by comparing couples of individuals of the population `pop`. The user must provide a `compare_function` that can receive the `r` field of two individuals and that returns `1` if the first individual is at least as good as the second, and returns `0` if not.

6.4 Scaling and sharing

`scale(pop,Cmult=2)` Performs linear scaling (Goldberg, 1989) over the population `pop`. The scaling constant `Cmult` has a default of 2.0.

`share(pop,sigma,type,alpha=1)` Performs sharing (Deb & Goldberg, 1991) of a given `type` with constants `sigma` and `alpha` over the population `pop`. The sharing type can be `'phenotypic'` or by `'fitness'`.

6.5 Methods that display and that sort a population

`sort(pop,type='phenotype',order='ascend')` Sorts a population `pop` according to a `type` of sort that can be by `'phenotype'` or `'fitness'` in and `order` that can be `'ascend'` (worst to best) or `'descend'` (best to worst).

`[x,f]=plot(pop,plotStyle='or')` Plots a population that has a single segment. The fitness of each individual is plotted versus its phenotype. Argument `plotStyle` is passed to Octave/MATLAB command `plot`. For example, the following commands produce the plot in figure 2:

```
>> p = population('r',[0 10],8);
>> p = random(p,40);
>> p = evaluate(p,@sqrt);
>> plot(p,'-o');
>> xlabel('x')
>> ylabel('f(x)')
```

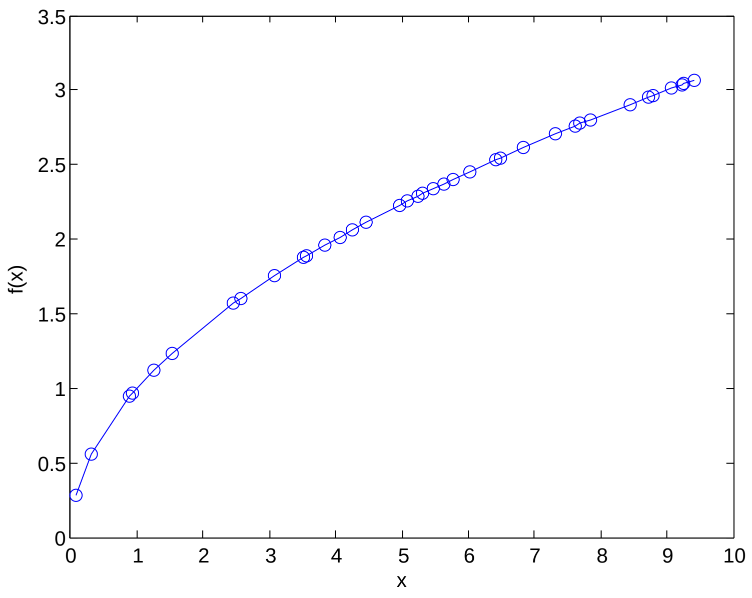


Figure 2: Example of the use of `plot`.

6.6 Mutation clock

In a traditional genetic algorithm, the mutation clock as suggested by Goldberg (1989) is the idea that instead of deciding if each individual bit is to be mutate, once a bit is mutate, the next bit to be mutate is determined using the appropriate probability distribution. There are (at least) two possible distribution probability for the random variable that gives the bit where the next mutation will occur. Using the geometric distribution, the next mutation occurs after

$$\left\lceil \frac{\ln\left(1-\mathrm{rand}()\right)}{\ln(1-p_m)} \right\rceil \tag{32}$$

bits, where p_m is the mutation probability per bit, and $\mathrm{rand}()$ returns a random number in $[0,1)$ with uniform distribution. Using an exponential distribution, the next mutation occurs after

$$\left\lceil -\frac{1}{p_m}\ln\left(1-\mathrm{rand}()\right) \right\rceil \tag{33}$$

bits[2]. The vgGA takes this idea and implements it over the value for traditional and generalized digits. vgGA2.0 defines mutation in terms of the expected number of mutations per segment, or m_s, which the user defines for each segment. Algorithms 1–5 define the way to calculate the value at which the next mutation happens, and the number of individuals to skip for the particular segment being processed. In these algorithms, the following variables are used:

p_m	Mutation probability.
m_s	Expected number of mutations per segment.

[2]Deb and Deb (2012, section 4.2) present this formula with a small typo.

m^+ Raw number of digits for next mutation.

n^+ Raw value for next mutation.

Δi Number of segments to skip for next mutation.

n Actual value where mutation falls.

k Real value where mutation falls.

Algorithm 1: Traditional digits, exponential distribution

```
1 $p_m \leftarrow \frac{m_s}{N - \log_B \delta + 1}$
2 $\phi \leftarrow \frac{\ln(1 - \text{rand}())}{\ln(1 - p_m)}$    % geometrical approximation
3 $m^+ \leftarrow m + \phi$
4 $\Delta i \leftarrow \left\lfloor \frac{m^+}{N - \log_B \delta + 1} \right\rfloor$
5 $m \leftarrow \lfloor m^+ \bmod (N - \log_B \delta + 1) \rfloor + 1$
6 $n \leftarrow B^{m-1}$
7 $k \leftarrow n\Delta\bar{r} + r_{\min}$
```

Algorithm 2: Generalized digits, exponential distribution

```
1 $p_m \leftarrow \frac{m_s}{N - \log_B \delta + 1}$
2 $\phi \leftarrow -\frac{\ln(1 - \text{rand}())}{p_m}$    % exponential approximation
3 $m^+ \leftarrow m - 1 + \phi$
4 $\Delta i \leftarrow \left\lfloor \frac{m^+}{N - \log_B \delta + 1} \right\rfloor$
5 $m \leftarrow (m^+ \bmod (N - \log_B \delta + 1)) + 1$
6 $n \leftarrow \lfloor B^{m-1} \rfloor$
7 $k \leftarrow n\Delta\bar{r} + r_{\min}$
```

Algorithm 3: Traditional digits, uniform distribution

```
1 $p_m \leftarrow \frac{m_s}{B^N/\delta - 1}$
2 $\phi \leftarrow -\frac{\ln(1 - \text{rand}())}{p_m}$    % exponential approximation
3 $n^+ \leftarrow n - 1 + \phi$
4 $\Delta i \leftarrow \left\lfloor \frac{n^+}{B^N/\delta - 1} \right\rfloor$
5 $n \leftarrow \left\lfloor \left(n^+ \bmod (B^N/\delta - 1)\right) + 1 \right\rfloor$
6 $m \leftarrow \lfloor \log_B n \rfloor + 1$
7 $n \leftarrow B^{m-1}$
8 $k \leftarrow n\Delta\bar{r} + r_{\min}$
```

Algorithm 4: Generalized digits, uniform distribution

```
1 $p_m \leftarrow \dfrac{m_s}{B^N/\delta - 1}$
2 $\phi \leftarrow -\dfrac{\ln(1-\text{rand}())}{p_m}$    % exponential approximation
3 $n^+ \leftarrow n - 1 + \phi$
4 $\Delta i \leftarrow \left\lfloor \dfrac{n^+}{B^N/\delta - 1} \right\rfloor$
5 $n \leftarrow \left\lfloor \left(n^+ \bmod (B^N/\delta - 1)\right) + 1 \right\rfloor$
6 $m \leftarrow \lfloor \log_B n \rfloor + 1$
7 $k \leftarrow n\Delta\bar{r} + r_{\min}$
```

Algorithm 5: Continuous digits, uniform distribution

```
1 $p_m \leftarrow \dfrac{m_s}{r_{\max} - r_{\min}}$
2 $\phi \leftarrow -\dfrac{\ln(1-\text{rand}())}{p_m}$    % exponential approximation
3 $k^+ \leftarrow k - r_{\min} + \phi$
4 $\Delta i \leftarrow \left\lfloor \dfrac{k^+}{r_{\max} - r_{\min}} \right\rfloor$
5 $k \leftarrow (k^+ \bmod (r_{\max} - r_{\min})) + r_{\min}$
```

The following methods implement these ideas:

`startMut(pop)` Initializes o reintializes the mutation clock.

`nextMut(pop,i)` Determines when the next mutation for segment `i` should occur accorinding to the mutation clock. Implements algorithms 1 through 5.

7 Full genetic algorithm

`[pop,data] = runGA(pop,methods,objective_function,generations)` This method can be used to implement a generational genetic algorithm. The population is initially evaluated using the `objective_function`. Then, the provided `methods` (plus `evaluate`) are applied to the population, for the given number of `generations`.

For example, the following commands create a population with two segments of 4 bits, fills the population with 20 random individuals, and runs the cycle of `tournament`, `crossover`, `mutation`, and `evaluate` for 20 `generations`. A plot of the best-found-so-far is created.

```
>> p = population('integer',[5 5 5],1,0.9);
>> p = random(p,20);
>> methods = {@tournament,@crossover,@mutation};
>> [p,data] = runGA(p,methods,@x2y2,20);
>> plot(data(:,1),data(:,2),'o-')
>> xlabel('function evaluations')
>> ylabel('best-found-so-far')
```

plotGA Please notice that this is not a method, but rather a function. Its location is the root directory vgGA2.0/.

```
[x,prom,desv]=plotGA(n,popsize,objective_function,methods,generations,seed,
 'integer',N,ms,pc,B,delta,dist,digits)
```

```
[x,prom,desv]=plotGA(n,popsize,objective_function,methods,generations,seed,
'real',R,N,ms,pc,B,delta,dist,digits)
```

This function runs n times a generational GA using runGA to generate a best-found curve (average of best found plus/minus a standard deviation) vs. the number of function evaluations. Each run is done for the indicated number of generations. The population size is popsize. Each generation, the methods indicated (plus evaluate) are applied to the population. If not given, methods takes the value of tournament, crossover, mutation. seed is passed to function rng, and can be 'shuffle' (randomized seed), 'default' (as if you restarted Octave/MATLAB), a positive integer, or empty. See method population for the rest of the parameters.

For example, the following instruction apply a genetic algorithm to the maximization of the function

$$f(x, y) = x + y, \tag{34}$$

where x and y are 4-bit binary integers. The expected number of mutations per individual per segment are 1 for x and 1 for y. Crossover probability is 0.9. The population size is 20. The genetic algorithm is run 10 times for 30 generations.

```
>> plotGA(10,20,@sum,[],30,'random','integer',[4 4],1,0.9);
```

8 vgGA tutorial

This section presents a brief tutorial on the use of the vgGA. You can find the source code in the file Documentation/Tutorial.m. The vgGA is programmed using the object oriented programming facilities of Octave/MATLAB. The vgGA is based on the class population defined in the directory @population/. In Octave/MATLAB the methods of a class must be placed in a directory whose name must start with a @ and that has the same name as the class.

The constructor of a class must be a function with the same name as the class, and returns an instance of the class that has been build using the function class. The manner in which an object is displayed is determined by the method display that the user must define.

Octave/MATLAB only allows for fields of an object to be accessed in the methods of its class. It is usual to build specific methods to retrieve and modify fields of an object. The methods get and set where written for this purpose, however, they are limited to the fields that were needed for testing, and they are not very general.

In Octave/MATLAB, unless explicitly declared global, all variables are local to the functions where they are used. Methods of a class are functions that must receive as an argument the object that they will work on, and must return it for its assignment if the object is to be modified.

8.1 Creation of an empty population

The constructor method is used to create an instance of the class population. For instance, the following instruction creates a population of integers, with two segments (one of 6 binary digits, and another of 3 digits of base 5), with expected mutations per segment of 0.6 and 0.4, and a crossover probability of 0.9:

```
>> p = population('integer',[6 3],[0.6 0.4],0.9,[2 5])
p is a vgGA population for maximization:

 parameters:
     type: integer
        N: 6 3
       ms: 0.6 0.4       pm: 0.12 0.2
       pc: 0.9
        m: 2
        B: 2 5
    delta: 2 5
 dig/dist: traditional/exponential
   DeltaR: 1 1

 Best (max) first found at evaluation 0, after 0 evaluations:
  BEST: 000000 000   0    0 ->        -Inf
```

This instruction creates an empty population (with no individuals).

8.2 Random individuals

The `random` creates 10 random individuals in the following instruction:

```
>> p = random(p,10)
p is a vgGA population for maximization:

 parameters:
     type: integer
        N: 6 3
       ms: 0.6 0.4       pm: 0.12 0.2
       pc: 0.9
        m: 2
        B: 2 5
    delta: 2 5
 dig/dist: traditional/exponential
   DeltaR: 1 1

 individual:
     1: 001000 424   8 114 ->         NaN
     2: 101000 022  40  12 ->         NaN
     3: 010001 233  17  68 ->         NaN
     4: 111101 440  61 120 ->         NaN
     5: 001010 441  10 121 ->         NaN
     6: 111101 220  61  60 ->         NaN
     7: 110011 032  51  17 ->         NaN
     8: 011010 424  26 114 ->         NaN
     9: 110010 434   50 119 ->         NaN
    10: 101001 004   41   4 ->         NaN

  average:       NaN median:       NaN std dev:       NaN
      min:       NaN    max:       NaN    skew:       NaN

 Best (max) first found at evaluation 0, after 0 evaluations:
  BEST: 000000 000    0   0 ->        -Inf
```

The genotype, phenotype, and evaluation of each individual is displayed. For example, individual 4 chromosome is `111101 440`, its phenotype is `[61 120]`, and its evaluation is `NaN` (*Not a Number*) because it has not been evaluated yet.

8.3 Evaluation

The method `evaluate` receives a function handle to an objective function and evaluates the population with it:

```
>> p = evaluate(p,@x2y2)
p is a vgGA population for maximization:

 parameters:
     type: integer
        N: 6 3
       ms: 0.6 0.4       pm: 0.12 0.2
       pc: 0.9
        m: 2
        B: 2 5
    delta: 2 5
 dig/dist: traditional/exponential
   DeltaR: 1 1

 individual:
     1: 001000 424    8 114 ->   1.306e+04
     2: 101000 022   40  12 ->        1744
     3: 010001 233   17  68 ->        4913
     4: 111101 440   61 120 ->   1.812e+04
     5: 001010 441   10 121 ->   1.474e+04
     6: 111101 220   61  60 ->        7321
     7: 110011 032   51  17 ->        2890
     8: 011010 424   26 114 ->   1.367e+04
     9: 110010 434   50 119 ->   1.666e+04
    10: 101001 004   41   4 ->        1697

  average:      9482 median: 1.019e+04 std dev:      6444
      min:      1697    max: 1.812e+04    skew:   -0.0208

 Best (max) first found at evaluation 4, after 10 evaluations:
  BEST: 111101 440   61 120 ->   1.812e+04
```

Individual 4 now has an evaluation of 1.812×10^4, and is the best found so far. `display` shows general information of the population: average fitness, best found so far, number of function evaluations, number of function evaluations to find the best, etc. In this example, the population is evaluated with function `x2y2` that returns the sum of the squares of the segments. For individual 4, the evaluation is $61^2 + 120^2 = 18120$. `evaluate` expects a function that can receive a vector which corresponds to the phenotype (its field `r`) of each individual and returns its evaluation.

8.4 Selection

Roulette wheel selection is implemented by the `roulette` method. (Other selection methods are also available (`tournament`, `sus`, `linRanking`) see section 6.3):

```
>> p = roulette(p)
p is a vgGA population for maximization:

 parameters:
     type: integer
        N: 6 3
       ms: 0.6 0.4       pm: 0.12 0.2
       pc: 0.9
        m: 2
        B: 2 5
    delta: 2 5
 dig/dist: traditional/exponential
   DeltaR: 1 1

 individual:
     1: 110010 434  50 119 ->   1.666e+04      1.757
     2: 110010 434  50 119 ->   1.666e+04      1.757
     3: 011010 424  26 114 ->   1.367e+04      1.442
     4: 011010 424  26 114 ->   1.367e+04      1.442
     5: 011010 424  26 114 ->   1.367e+04      1.442
     6: 111101 440  61 120 ->   1.812e+04      1.911
     7: 110011 032  51  17 ->        2890     0.3048
     8: 010001 233  17  68 ->        4913     0.5181
     9: 011010 424  26 114 ->   1.367e+04      1.442
    10: 001000 424   8 114 ->   1.306e+04      1.377

  average:  1.27e+04 median: 1.367e+04 std dev:      4962
      min:      2890    max: 1.812e+04    skew:    -1.082

 Best (max) first found at evaluation 4, after 10 evaluations:
  BEST: 111101 440  61 120 ->   1.812e+04



p is a vgGA population for maximization:
 parameters:
     type: integer
        N: 6 3
       ms: 0.6 0.4       pm: 0.12 0.2
       pc: 0.9
        m: 2
        B: 2 5
    delta: 2 5
 dig/dist: traditional/exponential
   DeltaR: 1 1

 individual:
     1: 110010 434  50 119 ->   1.666e+04      1.757
     2: 110010 434  50 119 ->   1.666e+04      1.757
     3: 011010 424  26 114 ->   1.367e+04      1.442
     4: 011010 424  26 114 ->   1.367e+04      1.442
     5: 011010 424  26 114 ->   1.367e+04      1.442
     6: 111101 440  61 120 ->   1.812e+04      1.911
     7: 110011 032  51  17 ->        2890     0.3048
     8: 010001 233  17  68 ->        4913     0.5181
     9: 011010 424  26 114 ->   1.367e+04      1.442
    10: 001000 424   8 114 ->   1.306e+04      1.377

  average:  1.27e+04 median: 1.367e+04 std dev:      4962
      min:      2890    max: 1.812e+04    skew:    -1.082

 Best (max) first found at evaluation 4, after 10 evaluations:
  BEST: 111101 440  61 120 ->   1.812e+04
```

8.5 Crossover and mutation

Crossover and mutation are performed by the `crossover` and `mutation` methods:

```
>> p = crossover(p)
p is a vgGA population for maximization:

 parameters:
     type: integer
        N: 6 3
       ms: 0.6 0.4       pm: 0.12 0.2
       pc: 0.9
        m: 2
        B: 2 5
    delta: 2 5
 dig/dist: traditional/exponential
   DeltaR: 1 1
 individual:
     1: 011010 434  26 119 ->         NaN        NaN
     2: 010001 234  17  69 ->         NaN        NaN
     3: 110010 424  50 114 ->         NaN        NaN
     4: 111010 424  58 114 ->         NaN        NaN
     5: 011010 424  26 114 ->   1.367e+04      1.442
     6: 001000 420   8 110 ->         NaN        NaN
     7: 010011 032  19  17 ->         NaN        NaN
     8: 110010 433  50 118 ->         NaN        NaN
     9: 011010 424  26 114 ->   1.367e+04      1.442
    10: 111101 444  61 124 ->         NaN        NaN

  average:       NaN median:       NaN std dev:       NaN
      min: 1.367e+04    max: 1.367e+04    skew:       NaN

 Best (max) first found at evaluation 4, after 10 evaluations:
  BEST: 111101 440  61 120 ->   1.812e+04

>> p = mutation(p)
p is a vgGA population for maximization:

 parameters:
     type: integer
        N: 6 3
       ms: 0.6 0.4       pm: 0.12 0.2
       pc: 0.9
        m: 2
        B: 2 5
    delta: 2 5
 dig/dist: traditional/exponential
   DeltaR: 1 1

 individual:
     1: 011010 434  26 119 ->         NaN        NaN
     2: 010001 234  17  69 ->         NaN        NaN
     3: 110010 424  50 114 ->         NaN        NaN
     4: 110010 124  50  39 ->         NaN        NaN
     5: 011010 424  26 114 ->   1.367e+04      1.442
     6: 001100 424  12 114 ->         NaN        NaN
     7: 010011 102  19  27 ->         NaN        NaN
     8: 110010 433  50 118 ->         NaN        NaN
     9: 011010 424  26 114 ->   1.367e+04      1.442
    10: 111101 144  61  49 ->         NaN        NaN

  average:       NaN median:       NaN std dev:       NaN
      min: 1.367e+04    max: 1.367e+04    skew:       NaN

 Best (max) first found at evaluation 4, after 10 evaluations:
  BEST: 111101 440  61 120 ->   1.812e+04
```

Notice that most individuals have an evaluation of `NaN` because they have not yet been evaluated. Individual 2 has an evaluation of 1429.000 because it was not modified by crossover or mutation.

8.6 Genetic algorithm

Methods `population`, `random`, `evaluate`, `tournament`, `crossover`, and `mutation` can be used by the `runGA` method to build a complete genetic algorithm. The following instructions run a population `p` for 50 generations, and the best found so far is plotted vs. number of function evaluations:

```
>> methods = {@tournament,@crossover,@mutation};
>> [p,B] = runGA(p,methods,@x2y2,50);
 gen  evals  best
   1     26  f(61,120) = 18121.000000
   2     32  f(61,120) = 18121.000000
   3     39  f(61,120) = 18121.000000
   4     44  f(61,120) = 18121.000000
   5     48  f(61,120) = 18121.000000
   6     56  f(61,120) = 18121.000000
   7     63  f(61,120) = 18121.000000
   8     70  f(61,120) = 18121.000000
   9     78  f(61,120) = 18121.000000
  10     87  f(61,120) = 18121.000000
  11     96  f(61,120) = 18121.000000
  12    102  f(61,120) = 18121.000000
  13    110  f(61,120) = 18121.000000
  14    118  f(61,120) = 18121.000000
  15    124  f(61,120) = 18121.000000
  16    131  f(61,120) = 18121.000000
  17    136  f(62,121) = 18485.000000
  18    142  f(62,121) = 18485.000000
  19    148  f(62,121) = 18485.000000
  20    154  f(62,121) = 18485.000000
  21    160  f(62,121) = 18485.000000
  22    168  f(61,122) = 18605.000000
  23    175  f(61,123) = 18850.000000
  24    181  f(61,123) = 18850.000000
  25    189  f(61,123) = 18850.000000
  26    198  f(63,123) = 19098.000000
  27    204  f(63,123) = 19098.000000
  28    211  f(63,123) = 19098.000000
  29    215  f(63,123) = 19098.000000
  30    223  f(63,123) = 19098.000000
  31    230  f(63,123) = 19098.000000
  32    237  f(63,123) = 19098.000000
  33    243  f(63,123) = 19098.000000
  34    249  f(63,123) = 19098.000000
  35    254  f(63,124) = 19345.000000
  36    260  f(63,124) = 19345.000000
  37    267  f(63,124) = 19345.000000
  38    272  f(63,124) = 19345.000000
  39    278  f(63,124) = 19345.000000
  40    287  f(63,124) = 19345.000000
  41    293  f(63,124) = 19345.000000
  42    299  f(63,124) = 19345.000000
  43    306  f(63,124) = 19345.000000
  44    312  f(63,124) = 19345.000000
  45    319  f(63,124) = 19345.000000
  46    327  f(63,124) = 19345.000000
  47    335  f(63,124) = 19345.000000
  48    342  f(63,124) = 19345.000000
  49    351  f(63,124) = 19345.000000
  50    358  f(63,124) = 19345.000000
```

```
>> plot(B(:,1),B(:,2))
```

The last instruction produce the plot in figure 3(a).

8.7 Best found plot

The `plotGA` function creates a population and repeatedly calls `runGA` to create a plot of the average best found, plus/minus a standard deviation for a series of genetic algorithm runs. The following instruciton runs a genetic algorithm 20 times for 50 generations.

```
>> plotGA(20,10,@x2y2,methods,50,'shuffle','integer',[6 3],[0.6 0.4],0.9,[2 5]);
```

Figure 3(b) shows the best found plot produced.

8.8 Objective functions

The following objective functions are implemented in the root directory of the vgGA and can be used for testing.

`debf1` Implements Deb's function F1 (Deb & Goldberg, 1991) used to test sharing.

$$f_1(x) = \sin^6(5\pi x) \tag{35}$$

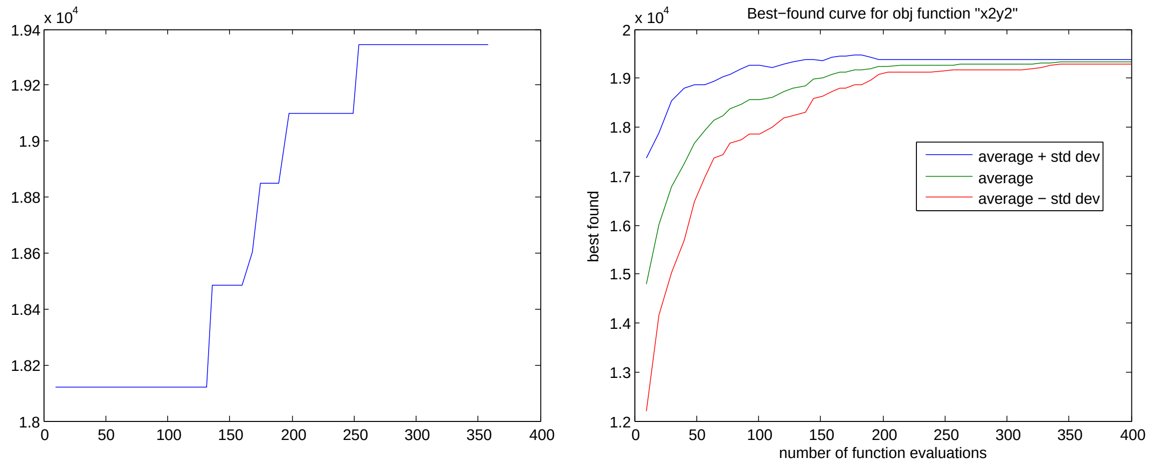
`debf2` Implements Deb's function F2 (Deb & Goldberg, 1991) used to test sharing.

$$f_2(x) = \exp\left(-2\ln 2\left(\frac{x-0.1}{0.8}\right)^2\right)\sin^6(5\pi x) \tag{36}$$

`onemax` Returns the number of ones in a chromosome.

`basemax(x,B=2)` For a chromosome in base `B`, returns the number of digits that are equal to $\texttt{B}-1$.

`x2y2` Returns the sum of the squares of the segments.



(a) Plot of the best found vs. number of function evaluations for the example in the tutorial.

(b) Plot of the best found for 20 runs of the genetic algorithm in this tutorial.

Figure 3: Plots create in the vgGA tutorial.

9 Using the vgGA with scripts

Instead of using the `runGA` method and the `plotGA` function, it is possible, and perhaps more convenient, to write a script that call the methods that one wishes to apply. Directory `Documentation/` contains example scripts that implement different algorithms:

`SimpleGA` This script implements a simple GA (selection, crossover, mutation) with tournament selection over the `basemax` function, obtaining the best found curve (average plus/minus a standard deviation).

`Scaling` This script shows the need for scaling in order to avoid diminished selective pressure at the end of a run. A genetic algorithm with proportional selection (SUS) is used to optimize Deb's function F2 shifted in the y axis by 20:

$$f_{2T}(x) = \exp\left(-2\ln 2\left(\frac{x-0.1}{0.8}\right)^2\right)\sin^6(5\pi x) + 20 \tag{37}$$

Figure 4 presents the distribution of the population over the objective function after 100 generations when no scaling is used. Figures 5(a) and 5(b) show the result after 50 generations when linear scaling and tournament selection is used. Without some kind of scaling, the population does not converge to the optimal of the objective function.

`Deception` This script uses functions `deceptive` and `interDeceptive` to test a genetic algorithm with tournament selection and crossover (no mutation) on problems formed by completely deceptive concatenated and interlaced problems. Figure 6 shows the average (plus/minus a standard deviation) of 30 runs of a genetic algorithm with a population of 400 individuals on a problem formed by 5-bit completely deceptive 10 subproblems

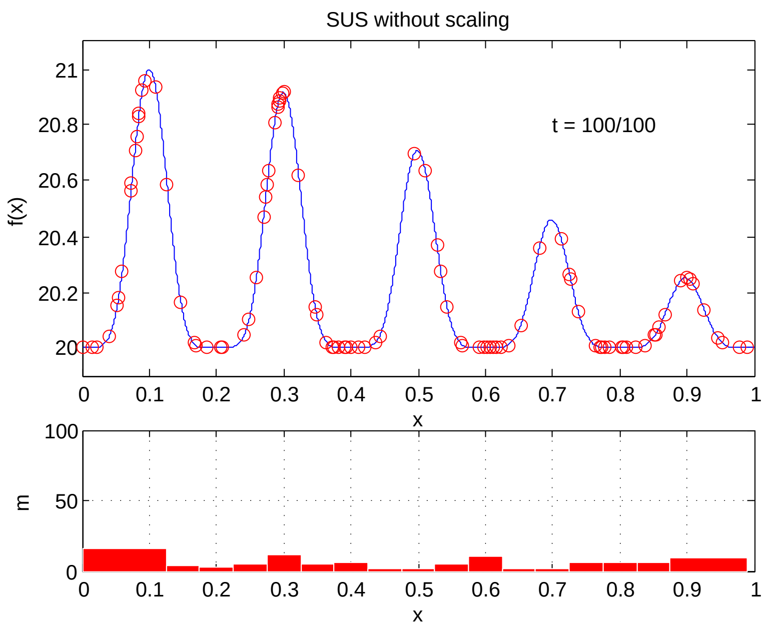
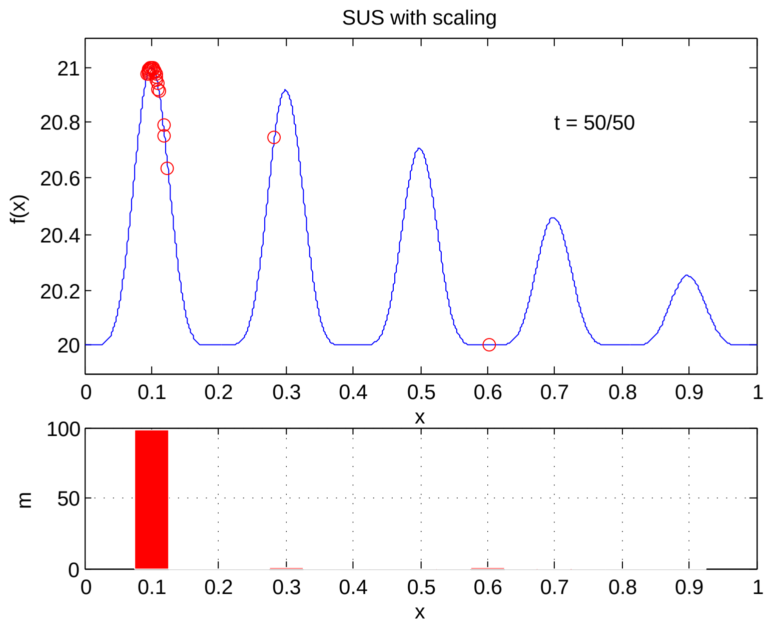


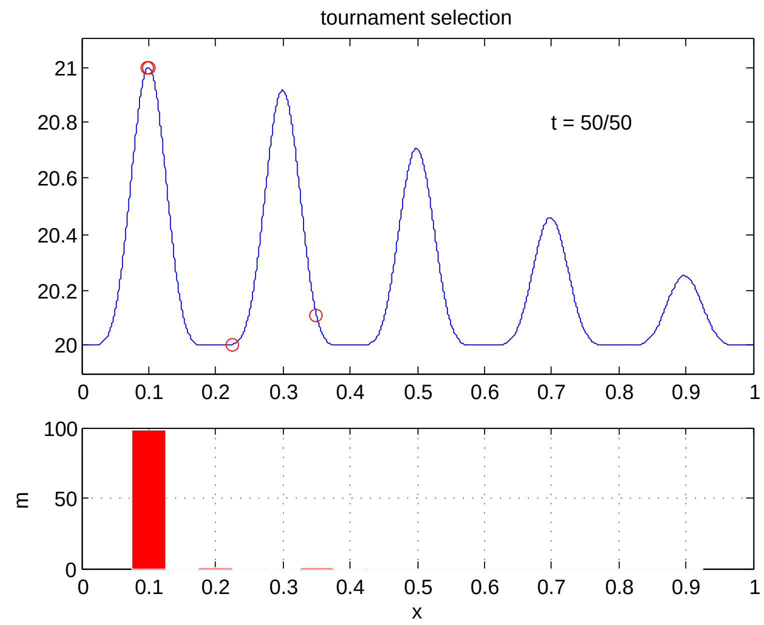
Figure 4: Simple genetic algorithm without scaling over translated Deb's function F1. Even after 100 generations, the population does not converge to the optimal at 0.1.

`Sharing` This is script a brief demonstration of sharing. A genetic algorithm with SUS selection and crossover is applied to Deb's functions F1 and F2 with, and without, sharing. Figure 7 shows the population after 200 generations.

`SteadyStateGA` This script is a rather crude comparison of a generational genetic algorithm (tournament selection, crossover, and mutation) and a steady-state genetic algorithm (roulette wheel, crossover, and mutation) optimizing the 50-bit `onemax` problem. This script shows how a steady-state GA can be implemented using `evaluateLast` and `eraseWeak` methods, and using the options `add`, `lastTwo`, and `last`, in `roulette`, `crossover`, and `mutation` methods. Figure 8 shows the result of a typical run of both algorithms.

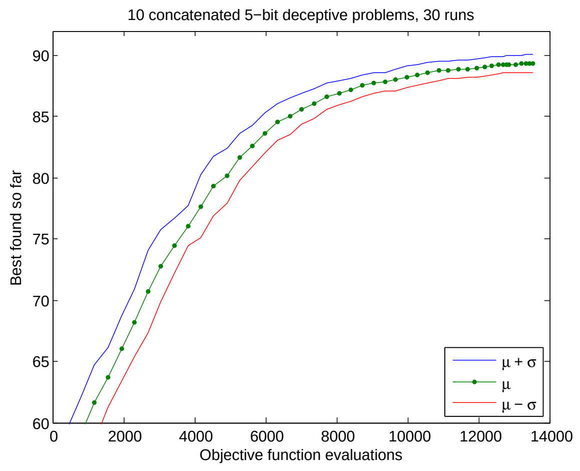


(a) Simple genetic algorithm with linear scaling over translated Deb's function F2. With scaling the population converges to the optimal.

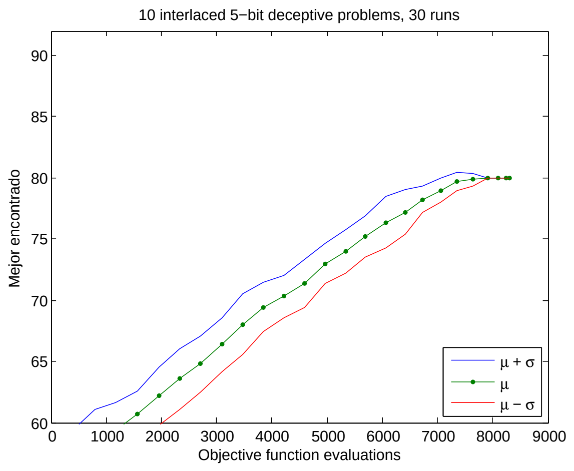


(b) Simple genetic algorithm with tournament selection over translated Deb's function F2. With tournament selection the population converges to the optimal.

Figure 5: With some for of scaling, a genetic algorithm can find the optimal of Deb's function F2 $+ 20$.

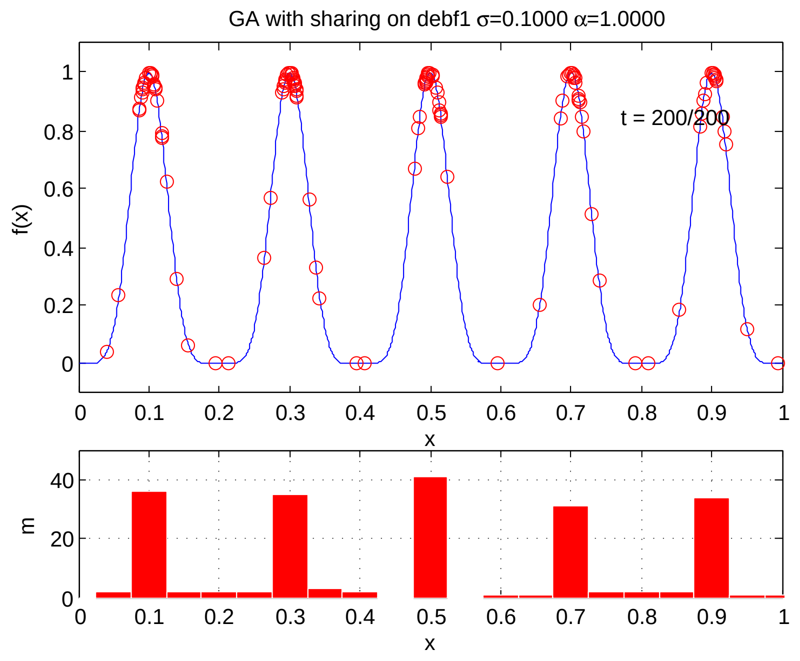


(a) Concatenated deceptive problems

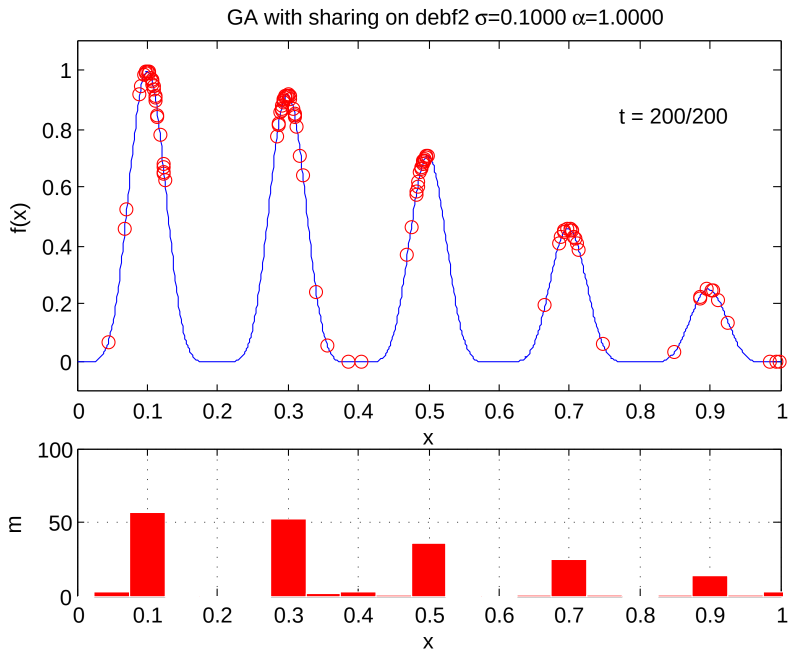


(b) Interlaced deceptive problems

Figure 6: Genetic algorithm over deceptive problems. If the problems are concatenated, the optimal is generally found. If the problems are interlaced, the optimal is never reached.



(a) Sharing on F1



(b) Sharing on F2

Figure 7: Demonstration of sharing over Deb's functions. The population distributes proportionally over the peaks of the objective function.

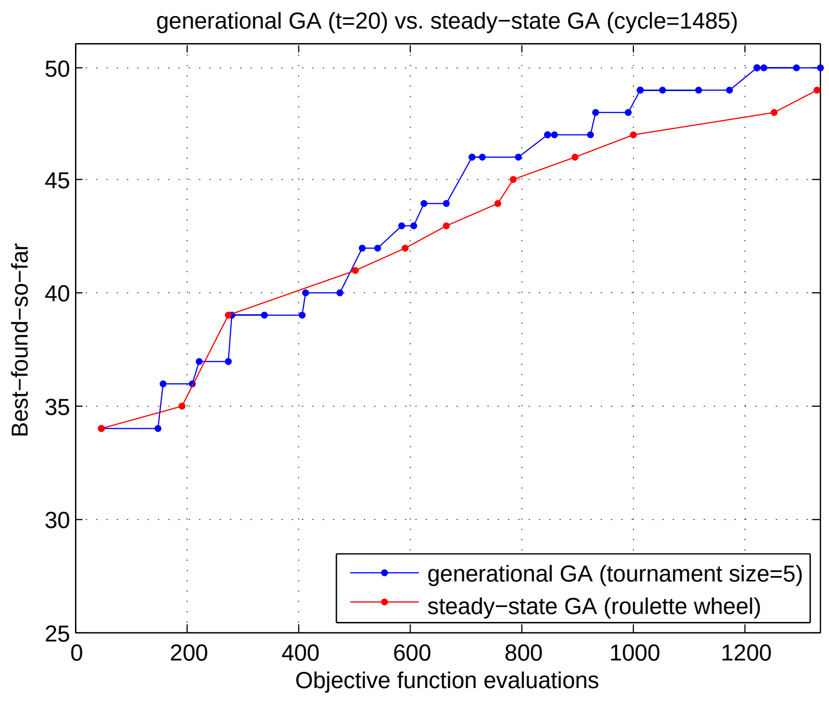


Figure 8: Comparison of a steady-state and a generational genetic algorithm on the 50-bit onemax problem. This is a typical run.

References

Baker, J. E. (1987). Reducing bias and inefficiency in the selection algorithm. *Proceedings of the Second International Conference on Genetic Algorithms and their Applications (ICGA-1987)*, 14–21.

Deb, K., & Deb, D. (2012). *Analyzing mutation schemes for real-parameter genetic algorithms* (KanGAL Report No. 2012016). Kanpur, India: Indian Institute of Technology.

Deb, K., & Goldberg, D. E. (1991). An investigation of niche and species formation in genetic function optimization. *Proceedings of the Third International Conference on Genetic Algorithms*, 42–50.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning*. Reading, MA: Addison-Wesley.

Goldberg, D. E., & Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. In G. J. E. Rawlins (Ed.), *Foundations of genetic algorithms (FOGA)* (pp. 69–93). San Mateo, CA: Morgan Kaufmann.

Spears, W. M., & De Jong, K. A. (1991). An analysis of multi-point crossover. In G. J. E. Rawlins (Ed.), *Foundations of genetic algorithms* (pp. 301–315). San Mateo, CA: Morgan Kaufmann.

Valenzuela-Rendón, M. (2003). The virtual gene genetic algorithm. *Genetic and Evolutionary Computation Conference (GECCO-2003)*, 1457–1468.

A Structure of population

The following is a listing of the file `@population/structures.m`:

```
evals        : number of function evaluations so far

params
          max: 0, 1 maximization flag
         type: 'integer', 'real'
            m: number of segments
         N[m]: digits per segment
      rMin[m]: min value of segments
      rMax[m]: max value of segments
    DeltaR[m]: (rMax-rMin)./B.^N
        ms[m]: expected number of mutations per segment
        pm[m]: mutation probability
           pc: crossover probability
         B[m]: base of segments
     delta[m]: mutation delta (width)
         dist: 'exponential', 'uniform'
       digits: 'traditional', 'generalized', 'continuous'
       dTitle: Display title?
       dIndiv: Display individuals?
       dTrace: Display tracing information?
         dMut: Display mutation clock?
       dStats: Display population statistics?
        dBest: Display best individual found so far?
      dChroms: Display chromosomes?
      dPhenos: Display phenotypes?
      dParams: Display population parameters?

best         : best individual found so far
         r[m]: phenotype
      fitness: evaluation
        evals: number of evaluation when found


individual[<population size>]
         r[m]: phenotype
      fitness: evaluation

mutclock
     nPlus[m]: value where next mutation will occur (integers)
     kPlus[m]: value where next mutation will occur (reals)
     mPlus[m]: digit where next mutation will occur
    DeltaI[m]: individuals to next mutation
      mMax[m]: maximum digit for mutation

trace
        nMuts: number of mutations
       nCross: number of crossovers
         nISC: number of Inter-Segment Crossovers
```

B Index of scripts, methods, functions, and fields