GEATbx Tutorial

Genetic and Evolutionary Algorithm Toolbox for use with Matlab

version 3.7 (November 2005)

Hartmut Pohlheim

Documentation for:

GEATbx version 3.7
(Genetic and Evolutionary Algorithm Toolbox for use with Matlab)

WWW: http://www.geatbx.com/
Email: support@geatbx.com

Contents

List of Figures

List of Tables

1 Introduction

The GEATbx (*Genetic and Evolutionary Algorithm Toolbox for use with Matlab*) contains a broad range of tools for solving real-world optimization problems. They not only cover pure optimization, but also the preparation of the problem to be solved, the visualization of the optimization process, the reporting and saving of results, and as well as some other special tools.

This Tutorial provides an introduction to the main GEATbx functions. The steps necessary for the efficient application of the GEATbx are explained. Nevertheless, this tutorial does not and cannot cover all the functions and aspects of the GEATbx. (Remember, you can always refer to the implemented code.)

The first steps for a 'Quick Start' are described in Chapter 2, p. and some examples are given. You can try these examples immediately and see the results seconds later. The examples use some of the GEATbx demos, giving a head start to those eager to try out the GEATbx.

From there on you have at least two ways of proceeding with this tutorial.

When using the GEATbx you need to know how to implement your problem ('Writing Objective Functions'). The procedure for doing so is described in Chapter 3, p.. Another important aspect which must be considered is the format of the 'Variable Representation', see Chapter 4, p..

The structure of the GEATbx is described in 'Overview of GEA Toolbox Structure' in an explanation of the 'Calling Tree' of the functions, see Chapter 5, p.. A brief overview of the interconnection between the GEATbx functions is also provided in this chapter. The GEATbx functions follow a 'Naming Convention', see Section 5.1, p..

The 'Data Structures of the GEATbx' are documented in Chapter 6, p.29.

All the direct algorithm documentation is done inside the Matlab m-files (`help name_of_m_file`). An extensive help text is provided for each function explaining the purpose and syntax and including illustrative examples. The M-function index (only in the on-line documentation) contains this information and additionally the dependencies between the functions (which function calls which other functions).

Years of work using the GEATbx to solve real-world problems has shown us, amongst other things, that the approach to new optimization problems is always one of the most important aspects. Chapter 7, p., 'How to Approach new Optimization Problems', explains a number of tips and steps.

If any part of the documentation does not address your problem, you can always contact the technical support for the GEATbx by email (support@geatbx.com).

2 Quick Start

The Genetic and Evolutionary Algorithm Toolbox (GEATbx) provides a number of demos. All these functions are called `demo*.m` (for instance `demofun1`). The demo-functions provide ready-to-run examples and can be called directly after the installation. All necessary parameters are already set.

2.1 First demonstration

As a first example, run the first demonstration function in Matlab:

```
demofun1;
```

This demo defines a number of evolutionary algorithm options, starts the optimization, displays the EA options employed on the screen (see figure 2-1), optimizes the objective function `objfun1`, returns intermediate information during the optimization in the Matlab command window (see figure 2-2 and 2-4), and visualizes the results graphically every few generations (see figure 2-3). The output on your screen might be slightly different, but the examples below give you a general idea of the output visualization. Below the output figures some of the options inside `demofun1` are explained.

Fig. 2-1: Output in Matlab command window at start of optimization run (used options)

```
Evolutionary Optimization
Objective function:  objfun1       Date: 22-Dec-2005     Time: 22:12:66
   number of variables:   10
   boundaries of variables:        -512
                                    512
Evolutionary algorithm parameters:
   subpopulations =      4       individuals =     25   (at start per subpopulation)
   termination    1:  max. generations = 400;
   variable format =    0   (real values - phenotype == genotype)
   selection
         function = selsus
         pressure =    1.7
         gen. gap =    0.9
      reinsertion
            rate =      1
   recombination
         name = recdis  recdis  reclin  recdis
         rate =      1
   mutation
           name = mutreal
           rate =      1
          range =    0.1    0.01   0.001  0.0001
      precision =     16
   regional model
      migration
            rate =   0.1    interval =   20
   output
      results on screen every  5  generation
      grafical display of results every  10  generation
```

```
method = 111111
style  = 614143
```

Fig. 2-2: Status information displayed in command window during optimization (some lines removed)

```
Generation   f-Count     Obj. Function          Term: 1        Time: cpu/gen,  full
    5.           451             40519          [   1.25%]     (  0.00min  00:00:02)
   10.           888             30140          [   2.50%]     (  0.00min  00:00:03)
   15.          1331             12935          [   3.75%]     (  0.00min  00:00:06)
   20.          1776            8572.8          [   5.00%]     (  0.00min  00:00:06)
   25.          2217            4910.8          [   6.25%]     (  0.00min  00:00:08)
   30.          2660            3644.4          [   7.50%]     (  0.00min  00:00:09)
   35.          3100            2414.4          [   8.75%]     (  0.00min  00:00:11)
   40.          3540            1210.6          [  10.00%]     (  0.00min  00:00:11)
   45.          3979            678.95          [  11.25%]     (  0.00min  00:00:13)
   50.          4418            208.31          [  12.50%]     (  0.00min  00:00:14)

  200.         17636       1.1752e-005          [  50.00%]     (  0.00min  00:00:54)
  205.         18076       8.6381e-006          [  51.25%]     (  0.00min  00:00:56)
  210.         18516       8.6179e-006          [  52.50%]     (  0.00min  00:00:57)
  215.         18956       5.6262e-006          [  53.75%]     (  0.00min  00:00:58)
  220.         19396       4.2513e-006          [  55.00%]     (  0.00min  00:00:59)

  370.         32604       4.4879e-008          [  92.50%]     (  0.00min  00:01:38)
  375.         33044       3.9711e-008          [  93.75%]     (  0.00min  00:01:40)
  380.         33484       3.4701e-008          [  95.00%]     (  0.00min  00:01:41)
  385.         33928       3.2459e-008          [  96.25%]     (  0.00min  00:01:42)
  390.         34368       3.1356e-008          [  97.50%]     (  0.00min  00:01:43)
  395.         34808       1.4482e-008          [  98.75%]     (  0.00min  00:01:45)
  400.         35248       1.4482e-008          [ 100.00%]     (  0.00min  00:01:46)
```

Fig. 2-3: Graphical output during optimization

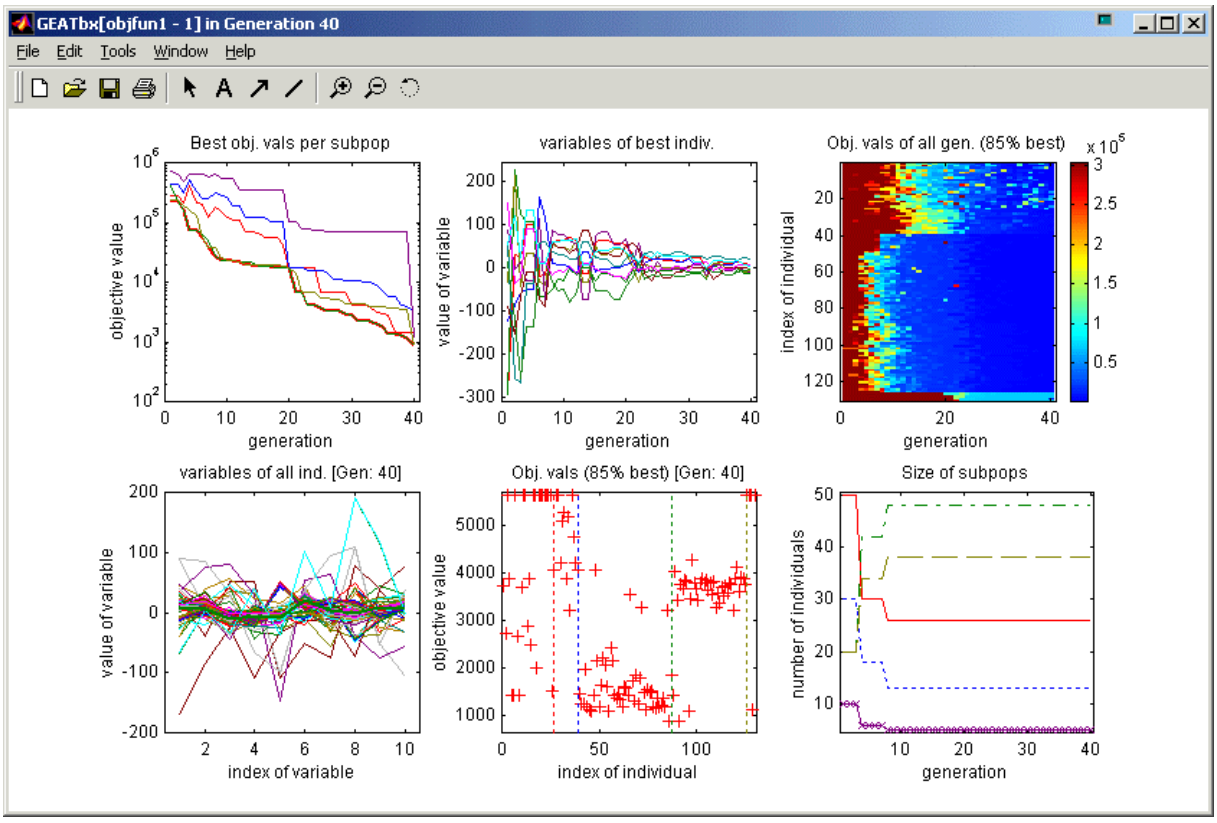


Fig. 2-4: Result information displayed in command window at the end of the optimization (some parts have been removed)

```
End of optimization: max. generations (400 generations; 1.78 time minutes)
Best Objective value:  1.44825e-008   in Generation 393
Best Individual:   -2.734e-005  -1.4079e-005   2.419e-005 -2.9976e-005
```

```
-2.8788e-005   9.4641e-005 -1.3272e-005  3.2009e-005
 2.7127e-005  -1.8188e-005
```

A number of options are defined in demofun1. Most of these options are not really necessary for a first demo function. However, you can use this function as a template. Let's have a look at some of the options.

The first step provides us with the default parameters for real valued parameters. More information can be found at Predefined Evolutionary Algorithms (in Parameter options).

```
% Get default parameters for real variables
GeaOpt = tbx3real;
```

The second step is to set number of subpopulations (here 5 subpopulations) and number of individuals per subpopulation (here a different number for each subpopulation, 50 individuals for the first subpopulation, 30 individuals in the second subpopulation and so on). Another useful parameter defines, how often textual status information is displayed during an optimization (here every 5 generations).

```
% Define special parameters
GeaOpt = geaoptset( GeaOpt ...
           , 'NumberSubpopulation',   5 ...
           , 'NumberIndividuals',   [50, 30, 20, 20, 10] ...
           , 'Output.TextInterval',   5 ...
          );
```

The definition of the objective function is quite important. Here we use one of the many example objective functions of the GEATbx, objfun1 implementing the hyper sphere function. The objective function is implemented in an m-function. Thus, we simply provide the name of this m-function:

```
% Define objective function to use
objfun = 'objfun1';
```

All the example objective functions of the GEATbx not only contain the definition of the function, but also all the necessary parameters for the application of the function: boundaries of the variables (VLUB: vector of lower and upper bounds), a short textual description, the best objective value (when known), and the corresponding variable values. Thus, the GEATbx can always "ask" the current objective function for the parameters needed. A gateway function to access these parameters is provided.

```
% Get variable boundaries from objective function
VLUB = compdiv('getdata_objfun', objfun, 1); % GEATbx v.3.3
VLUB = geaobjpara(objfun, 1); % GEATbx v.3.4 and newer
```

A full description of the gateway into the objective functions is given in 'Writing Objective Functions', Chapter 3, p..

Now all options are defined. Thus, the evolutionary algorithm implemented in geamain2 can be called:

```
% Start optimization
[xnew, GeaOpt] = geamain2(objfun, GeaOpt, VLUB);
```

During the optimization status information and visualization output are displayed on the screen (see figure 2-2, 2-4, and 2-3).

If you want to have a look at the search space of the objective function (visualizing the first 2 dimensions):

```
% Do a mesh plot of the objective function
plotmesh(objfun, [-100,-100;100,100]);
```

2.2 Second demonstration

In order to give an impression of the utility of the GEATbx, a demonstration implementing 3 different evolutionary algorithms and calling many of the example objective functions is provided. Call the demonstration in the Matlab command window:

```
demogeatbx(1,1);
```

This call optimizes the objective function objfun1 with a 'Globally oriented optimization (with multiple subpopulations)' nearly identical to the implementation in demofun1.

When the demonstration is called without parameters, you can select one of the available options from a text menu. The first menu provides a selection of at least 16 different objective functions. The second menu offers 3 different evolutionary algorithms (see figure 2-5, bottom).

Fig. 2-5: Demonstration demogeatbx: option selection in menu (top: objective function, bottom: evolutionary algorithm to apply)

```
» demogeatbx
--- Please select objective function to use ----
      1) Sphere function (objfun1)
      2) ROSENBROCKs function (objfun2)
      3) RASTRIGINs function (objfun6)
      4) SCHWEFELs function (objfun7)
      5) GRIEWANGKs function (objfun8)
      6) Sum of different power (objfun9)
      7) ACKLEYs path function (objfun10)
      8) LANGERMANNs function (objfun11)
      9) MICHALEWICZs function (objfun12)
      10) Axis parallel ellipsoid (objfun1a)
      11) Rotated ellipsoid (objfun1b)
      12) Moved axis parallel ellipsoid (objfun1c)
      13) Live optimization (objlive1, 2 dim)
      14) Live optimization (objlive1, 10 dim)
      15) FONSECAs MO function 1 (mobjfonseca1)
      16) FONSECAs MO function 2 (mobjfonseca2)
Select a menu number: 1

--- Please select the optimization method to use! ----
      1) Globally oriented optimization (multiple subpops)
      2) Globally oriented optimization (1 subpop)
      3) Locally oriented optimization
Select a menu number: 1
```

Using this demonstration function you can explore many of the provided objective functions. Each of the objective functions has different properties. Thus, for each objective function the 3 evolutionary algorithms behave differently. For instance, try the following combinations:

```
demogeatbx(1,2);     demogeatbx(1,3);
demogeatbx(2,3);     demogeatbx(2,1);
demogeatbx(12,2);
demogeatbx(5,1);     demogeatbx(5,3);
```

The options of the evolutionary algorithms employed are displayed in the Matlab command window at the beginning of the optimization. However, the first 2 evolutionary algorithms ('Globally oriented optimization', most options defined in `tbx3real`) differ only in the number of subpopulations employed. Both of these use 200 individuals. The third evolutionary algorithm ('Locally oriented optimization') uses an evolution strategy as the mutation operator and no recombination. It works with only 12 individuals. A few other options are also different. Most of the options of this evolutionary algorithm are set inside `tbx3es1`.

2.3 Your first optimization of an own objective function

Up to now you have only used the demonstration and objective functions provided. It is time to write your first objective function. Let's implement a simple quadratic function with a moved center.

An implementation of the `x^2` function with a moved minimum point to [4, 8, 12, ...] is given in figure 2-6. Save these lines in an m-file with the name `objexample1.m`. Now the objective function is defined.

Fig. 2-6: Definition of objective function objexample1

```
function ObjVal = objexample1(Chrom)
   % Compute population parameters
   [Nind, Nvar] = size(Chrom);
   % Move the center of the minimum
   Chrom = Chrom - 4 * repmat([1:Nvar], [Nind, 1]);
   % Calculate the objective values
   ObjVal = sum((Chrom .^2)')';
```

In the command window you can now set a few optimization options before you start the optimization (all other options are taken from the set of default options). These lines can also be written to an m-file creating your own demo example and executing this m-file.

As an example, set the interval of text output to every 3 generations. The second option results in an update of the graphical output every 9 generations. The optimization stops after half a minute (termination method 2 is maximal time in minutes). Using the new objective function and the special options you can call the evolutionary algorithm. Here we also provide the number of variables or dimensions of the objective function (4 variables by defining 4 lowerand 4 upper bounds) and the boundaries of these variables (-50 and 50).

Fig. 2-7: Definition of options, size of the problem (number of variables) and execution of optimization

```
% Define special parameters
GeaOpt = geaoptset( 'Output.TextInterval',    3 ...
                  , 'Output.GrafikInterval',  9 ...
                  , 'Termination.MaxTime',  0.5 ...
                  , 'Termination.Method',     2);
```

```
VLUB = [-50,-50,-50,-50; 50,50,50,50];
geamain2('objexample1', GeaOpt, VLUB)
```

Using these few lines the optimization is executed and results are shown in the command window (text output) and the graphical result figure. At the end of the run the result is given in the command window (similar to figure 2-4, p.).

The same objective function can easily be optimized for 15 variables/dimensions or more. To do so simply define a larger number of variable boundaries. Now the objective value can only reach a minimum of 140. The boundaries for the higher dimensions do not allow the (unbounded) minimal value to be found. You must therefore call the evolutionary algorithm again with extended/changed boundaries.

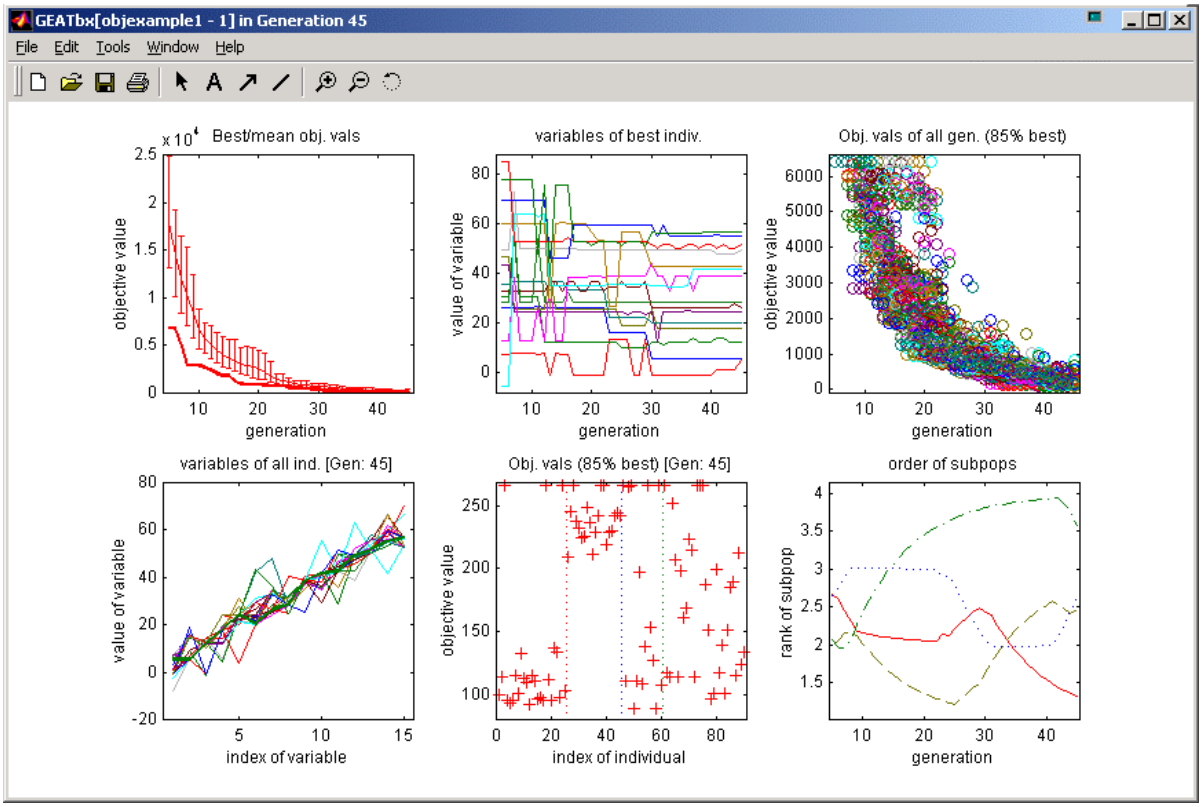
Fig. 2-8: Definition of a larger number of variables and with extended boundaries

```
% Define 15 variables
VLUB = repmat([-50; 50], [1, 15]);
geamain2('objexample1', GeaOpt, VLUB)

% Define different boundaries
VLUB = repmat([-12; 123], [1, 15]);
geamain2('objexample1', GeaOpt, VLUB)
```

An impression of the graphical output during optimization is given in figure 2-9.

Fig. 2-9: Graphical output during optimization of first own objective function



These are the graphics which default options provided. The top left graph presents the best objective value during the last 40 generations (and the standard deviation of all objective values). The top middle graph visualizes the variables of the best individual over the last 40 generations. Out

of the chaos at the beginning, an order seems to emerge and you can try to interpret the results. The top left graph shows all the objective values (to be exact only 85% of the best of each generation) over the last 40 generations. A very interesting graph is presented on the bottom left: all the variables of all the individuals of the current generation. You can clearly see the order in the variable values. The bottom middle graph shows the objective values of the individuals of the current generation and the bottom right graph the order of the subpopulations (both will be explained at a future date).

During optimization the graphical output changes quite substantially. Thus, the graphic in figure 2-9 is just a snapshot. If, however, you follow the changing display during optimization you will soon gain a good insight into the course and the state of the current optimization.

2.4 Further Steps

The examples shown will be sufficient for your first steps using the GEATbx. Have a look at the `scripts` subdirectory of your `geatbx3` installation. This directory contains further demo functions – see the respective help text for more information.

The next step is to learn about 'Writing Objective Functions', see Chapter 3, p., and how to handle the different 'Variable Representation' in the objective function and the evolutionary algorithm, see Chapter 4, p..

For an 'Overview of GEA Toolbox Structure' and more information on the 'Calling Tree' of the toolbox, see Chapter 5, p..

3 Writing Objective Functions

When using the toolbox the implementation of an objective function consumes most of the work. Inside this function the calculation of the objective values depending on the variables is performed. Here you must implement your problem! The kind of implementation determines how good the evolutionary algorithm can work on and solve the problem.

Included in the distributed version of the toolbox are many examples of objective functions. All included objective functions follow the naming convention obj*.m (see also Naming Convention, Section 5.1, p.). These functions implement a broad class of parameter optimization problems (and a few ordering/scheduling problems). When using these functions as a template it will be easier to implement your own functions/problems. For an overview of the mathematical description see *EXAMPLES OF OBJECTIVE FUNCTIONS*.

Consider the following tasks:

1. The objective function is called with a **matrix with as many rows as individuals**. Every row corresponds to one individual. The number of **columns determines the dimension**/the number of variables of the objective function (see also Data Structures of the GEATbx, Chapter 6, p.29).
2. Because of being called with many individuals the objective function calculates the same mathematical expressions more than once. This could be done in a for-loop. However, here is a highly recommended place for **vectorization**.
3. Beside calculating the objective values the objective function could be used for **defining default values** (boundaries: lower and upper bounds of the variables, a default dimension of the problem; additionally, a descriptive string (name of the objective function) for labeling plots and, if known, the minimal objective value; for multi-objective problems the number of objective values).

3.1 Parametric optimization functions

Let's finish the theory. Here is an example.

Consider implementing the simple quadratic function (sum of quadrate or bowl function; known as DE JONG's function 1 - `objfun1`). This functions works on real value variables.

Fig. 3-1: Definition of an objective function

```
function objval = objfun1(x)
   objval=sum((x.*x),2);   % vectorized, thus fast
   % for i=1:size(x,1), objval(i)=sum(x(i,:).^2); end
```

`objfun1` returns the objective values of all individuals and because of the vectorization it will be fast. The second line (commented) shows the implementation of this function unvectorized. Every individual will be computed separately. The result is the same, however, *Matlab* takes a considerably longer time.

A fully documented version of the above objective function is implemented in `objfun1`.

Other examples of objective functions (using real value variables, vectorized implementation and a definable number of dimensions) are:

- `objfun1a` (axis parallel hyper-ellipsoid)
- `objfun1b` (rotated hyper-ellipsoid)
- `objfun2` (ROSENBROCK's function)
- `objfun6` (RASTRIGIN's function)
- `objfun7` (SCHWEFEL's function)
- `objfun8` (GRIEWANGK's function)
- `objfun9` (sum of different power)
- `objfun10` (ACKLEY's path)
- `objfun11` (LANGERMANN's function)
- `objfun12` (MICHALEWICZ's function)

These functions are very often used as a standard set of test functions for evaluation of the performance of different evolutionary algorithms.

3.2 Defining default values of the objective function

The next step is defining default values for boundaries (domain of variables), best objective value and a description for the function. During the work on the toolbox the following style developed and is implemented inside all provided objective functions: Call the objective function with one special individual only (first variable is NaN, the optional second defines the type of info requested, for example x = [NaN, 1]).

Fig. 3-2: Definition of special return values of an objective function

```
function objval = objfun1(x, option)
   [Nind,Nvar]=size(x);
   % if Chrom is [NaN xxx] define size of boundary-matrix and others
   if all([Nind == 1, isnan(Chrom(1))]),
      % If only NaN is provided
      if length(Chrom) == 1, option = 1; else option = Chrom(2); end
      % Default dimension of objective function
      Dim = 20;
      % return text of title for graphic output
      if option == 2, ObjVal = ['DE JONGs function 1'];
      % return value of global minimum
      elseif option == 3, ObjVal = 0;
      % define size of boundary-matrix and values
      else
         % lower and upper bound, identical for all n variables
         ObjVal = repmat([-512; 512], [1 Dim]);
      end
   else
       % Compute objective function
   end
```

The line `objfun1([NaN,1])` or `objfun1([NaN])` will return the default boundaries (includes implicitly the dimension) of the objective function. This is used inside a number of functions to retrieve the default boundaries and implicitly the number of variables for a given objective function.

```
objfun='objfun1';
VLUB = feval(objfun, [NaN, 1]);
geamain2(objfun, [], VLUB);
```

Getting the descriptive name of the objective function (`objfun1([NaN,2])`) or the minimal objective value (`objfun1([NaN,3])`) is not used in the distributed version of the toolbox (because I can not guaranty that you write all your objective functions according to this scheme). However, when comparing different algorithms and defining a termination criterion against the minimal objective value it is very useful (and I use it quite often for my own work).

A fully documented version of the above objective function is implemented in `objfun1`.

3.3 Optimization of dynamic systems

Often the solution of a problem involves the simulation of a system or the call of other functions. Consider the optimization of the control vector of a double integrator (push cart system). An overview of this system is given in *EXAMPLES OF OBJECTIVE FUNCTIONS*.

```
function objval=objdopi(Chrom,option)
   [Nind,Nvar]=size(Chrom);
   XINIT=[0;-1];XEND=[0;0];
   TSTART=0; TEND=1;TIMEVEC=linspace(TSTART,TEND,Nvar)';
   if Nind==0,
   % see above example or objdopi
   else
      STEP=abs((TEND-TSTART)/(Nvar-1)));
      for indrun=1:Nind
         control=[TIMEVEC [Chrom(indrun,:)]'];
         [t x]=rk23('simdopiv', [TSTART TEND], XINIT, ...
                    [1e-3,STEP,STEP],control);
         % Calculate objective function
         objval(indrun)=sum(abs(x(size(x,1),:)'-XEND))+ ...
                        trapz(t,Chrom(indrun,:).^2));
      end
   end
```

At the beginning of the calculation of the objective values problem specific parameters are defined (`XINIT, XEND, TSTART, TEND, TIMEVEC`). The direct calculation is done separately for every individual (`rk23` is written for only one system at one time). Every individual is converted in the form required by `rk23` (vector of time values in the first column, next column(s) contain control values at specified time). The simulation function is called with appropriate parameters, the state values are returned. (`rk23` is a Matlab4 specific function. The calling and setup of the integration routine changed under Matlab5 - see help for `sim` and `simset`.)

The objective value is a combination of two terms:

- sum of all values of the individual (the needed energy/force to change the state of the system) and

- difference between reached and needed end value of the states.

The objective function is finished. However, there is still another function needed - the s-function for the simulation routine `simdopiv`. (for an introduction to writing s-functions see the *Simulink reference guide* or look at the provided s-functions of the GEATbx `sim*.m`).

```
function [sys,x0]=simdopiv(t,x,u,flag);
   if abs(flag) == 1
      sys(1,:) = u(1,:);
      sys(2,:) = x(1,:);
   elseif abs(flag)==0
      sys=[2,0,0,1,0,0]; x0 = [0; -1];
   end
```

Let's go one step further. The toolbox provides the possibility to pass up to 10 parameters to the objective function. In the above example it could be useful to have the chance of changing `TSTART, TEND, XINIT, XEND` from outside and thus optimizing different situations of the double integrator system. The beginning of the function would be changed to:

```
function objval=objdopi(Chrom,option,TSTART,TEND,XINIT,XEND)
   [Nind,Nvar]=size(Chrom);
   if nargin<3, TSTART=0; end
   if nargin<4, TEND=1; end
   if nargin<5, XINIT=[0;-1]; end
   if nargin<6, XEND=[0;0]; end
```

The problem specific parameters are checked and if not provided set to default values. Thus, if necessary the parameters can be passed to the function or default values will be used. For optimizing `objdopi` the script would be (implemented in `demodopi`):

```
objfun='objdopi';
TSTART=0; TEND=2; XINIT=[0;-2]; XEND=[0;0];
tbxmpga(objfun, [], [], 0, TSTART, TEND, XINIT, XEND)
```

An even more advanced parameter checking could be done with (example):

```
   if nargin<3, TSTART=[]; end
   if isempty(TSTART), TSTART=0; end
   if nargin<4, TEND=[]; end
   if isempty(TEND), TEND=1; end
```

The parameter passing mechanism offers the possibility of getting multiple solutions at once without editing the objective function:

```
   for i=1:10,
      XINIT=[0;-i];
      geamain2(objfun,GeaOpt,VLUB,[],0,[],[],XINIT)
   end
```

Undefined parameters (`TSTART`, `TEND`, `XEND`) will be set to default values - really useful in day to day work.

One problem remains - vectorization. Many simulation problems are not vectorizable and thus slow. The evolutionary algorithm spends most of the time calculating the objective values. For simulation functions there are actually two problems: The Matlab-provided integration functions (`rk23`, `rk45` or `sim`) are not vectorized. And it's often difficult to vectorize the s-function (see `sim*.m`). However, for this example both of these problems are solved. The s-function `simdopiv`

is vectorized and together with the GEATbx a vectorized integration routine, `intrk4`, is provided. For more information see the documentation of `intrk4` and method=12 inside `objdopi`.

If the objective function computes quite a few temporary results that are not part of the objective value it is good style to return them as additional output parameters.

```
function [objval, t, x]=objdopi(...)
```

What is the benefit? This opens the possibility of writing problem specific result plotting or special result computing routines. An example is implemented for the double integrator, which serves at the same time as an example for these features.

1. special initialization function (`initdopi`), [provides problem specific knowledge during initialization of the population].
2. special state plot function (`plotdopi`), [plot special results for best individual during optimization, for instance state and output variables of simulation].

For using these features the appropriate GeaOpt parameters (`Special.InitDo`, `Special.InitFunction`, `Output.StatePlotInterval`, `Output.StatePlotFunction`) must be defined. When calling the evolutionary algorithm `Special.InitDo` should be set to 1 and `Output.StatePlotInterval` should be set to 1 or greater. The corresponding *Function options contain the manes of the special initialization or state plot functions. The concept of setting the parameter in the options structure for using (or not) the special feature and defining the name of the special function is very flexible. For every objective function a special function for initialization or state plot can be defined (every problem needs it's own initialization or state plot function). Via parameter the use can be switched on or off. An example of all this is provided with the demo function `demodopi`, the objective function `objdopi`, the initialization function `initdopi`, and the state plot function `plotdopi`.

3.4 Remark

Two things should be stated at the end:

- Before starting to write own functions have a look to the provided example functions. It's much easier to use one of them as a template than starting from scratch. It is not necessary that everybody solves the same problems again and again.
- If good examples are known, which could be used for introducing a whole class of problems, please send them to the me (support@geatbx.com). If appropriate they will be included into the documentation of the toolbox and this tutorial as an example function.

4 Variable Representation

One important step in deciding which evolutionary algorithm is to use is a close look to the format/representation of your variables. A second step is the direct decision on which format the evolutionary algorithm should work. The representation determines the overall algorithm, that means, the used evolutionary operators.

In the GEATbx different representations are directly supported:

1. real value representation,
2. binary value representation,
3. integer value representation,
4. ordering representation.

When working with different representations, the toolbox provides functions for conversion between these representations:

- binary to integer (`bin2int`)
- binary to real (`bin2real`)

Let's give an example: The variables of the objective function are in real value representation. Now it could be chosen between binary and real value representation for the evolutionary algorithm. Recommended is the real value representation (`tbx3real`). It works much quicker than the binary one. However, if the decision comes to use the binary values inside the evolutionary algorithm, the population is initialized as if the variables would be binary (`initbp`), the evolutionary operators (`mutbin` and, for instance, `recsp`) are applied, and before the evaluation of the objective function the binary values are converted to real values (`bin2real`). The toolbox function `tbx3sga` defines all these options.

One more example. In this case the variables are in integer representation. The evolutionary algorithm can work on integer values (`mutint` and `recdis`) or with binary values and convert them to integer (`bin2int`) before evaluation of the objective function.

The use of a representation and necessary conversion is controlled by the parameter `VariableRepresentation`.

Tab. 4-1: Combinations of variable representation and conversion

`VariableRepresentation`	EA works on	variable representation	conversion
0	real	real	-
1	integer	integer	-
2	binary	real	`bin2real`
3	binary	integer	`bin2int`
4	binary	binary	-

In the end it can be stated:

- Use nearly always the “natural” representation of the variables!

- If the variables of the objective function are **real** use the **real value presentation** for the evolutionary algorithm as well.
- If the variables of the objective function are **binary** use the binary value presentation for the evolutionary algorithm as well.
- If the variables of the objective function are **integer** use the **integer value presentation** for the evolutionary algorithm.

Matlab Examples:

Real --- Real:

```
geamain2('objfun1')
```

- All default parameters of the GEATbx are defined for a real representation.
- `objfun1` is an objective function using real representation.

Binary --- Binary:

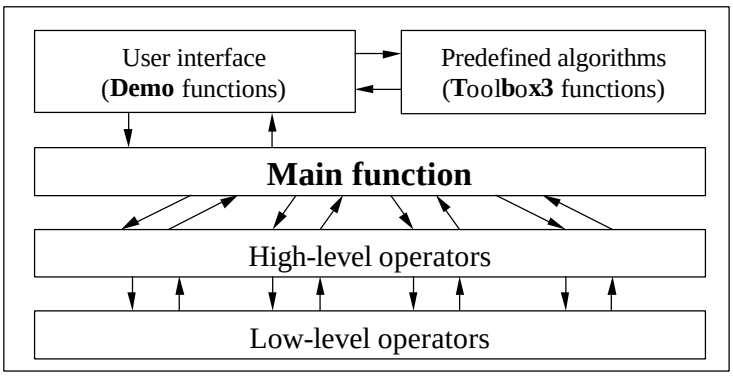
```
objfun = 'objone1'; GeaOpt = tbx3bin; VLUB = geaobjpara(objfun, 1);
geamain2(objfun, GeaOpt, VLUB)
```

- `objone1` is an objective function using binary representation
- `tbx3bin` defines default parameters for a binary representation used by the EA

5 Overview of GEA Toolbox Structure

The GEATbx is build around a layer model. Figure 5-1 gives a first overview of the layers used for the GEATbx.

Fig. 5-1: Layer model of the GEATbx



Central point is the Main function (`geamain2`). This function is called from the user interface (Demo functions, for instance, `demofun1`, `demogeatbx`). The Toolbox functions define default parameters for a number of different evolutionary algorithms. The Main function calls all necessary evolutionary operators. The call of evolutionary operators is done by a high-level layer calling the low-level layer. The Main function also calls the objective function. Additionally, the Main function performs nearly all the data management and result collection.

Figure 5-2 shows the full structure of the GEATbx using the names of the corresponding m-functions. Even in this detailed view the layer model is visible. Center of the toolbox is the Main function `geamain2`. The Main function is called by the Demo functions. The Demo functions get parameter sets of predefined algorithms via the Toolbox functions (***tbx3****, `tbx3real`, `tbx3bin`, `tbx3es1`). The Main function calls the high-level operators (`selection`, `recombin`, `mutate`). The high-level operators call the low-level operators (`selsus`, `recdis`, `mutreal`). This layer model not only provides a good overview of the structure of the GEATbx. It makes the extension of the toolbox by new operators or special functions straightforward.

In this structure there are two main points of user input; the objective function and the demo function.. On the one hand the user must provide an objective function implementing its problem to solve. This task is described in Chapter 3, p.11. On the other hand the user must define the evolutionary algorithm to employ and define none, some or many of the provided options of the GEATbx. This can done in one central place. All the provided examples of the GEATbx are called demo functions and can be used as a starting point.

5.1 Naming Convention

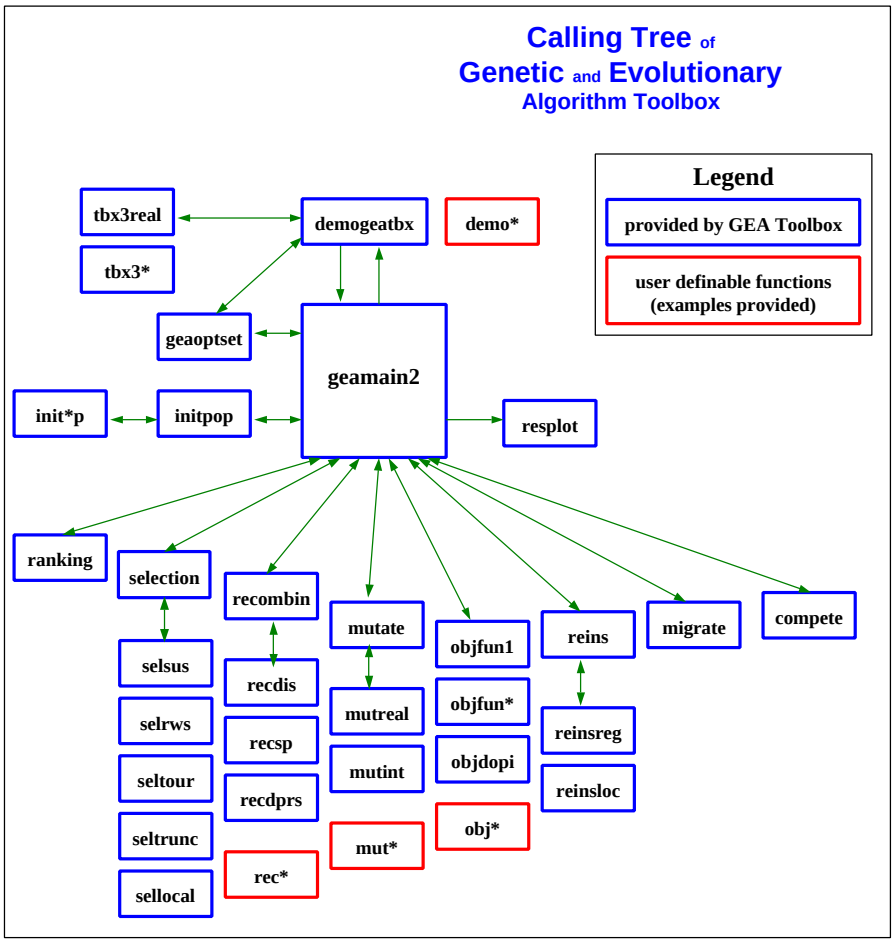
Throughout the GEATbx the functions are named according to the following conventions.

Tab. 5-1: Naming convention of the GEATbx

Präfix	group of functions or operators
demo	high-level **demo** functions (set of options for a special problem, often used as demonstration)
tbx3	high-level **t**ool**b**o**x** algorithms (define a special evolutionary algorithm of the toolbox)
init	**init**ialization functions
sel	low-level **sel**ection operators
rec	low-level **rec**ombination operators
mut	low-level **mut**ation operators
reins	low-level **reins**ertion operators
obj	**obj**ective functions (implement a special problem)
mobj	**m**ulti-**obj**ective functions
sim	**sim**ulation function (called by corresponding objective function)
plot	application specific **plot** functions

5.2 Calling Tree

Fig. 5-2: Calling tree of the Genetic and Evolutionary Algorithm Toolbox (GEATbx)



5.3 Demo / Startup function

At the beginning there is a high-level function for defining some parameters and the application specific details. These functions are called demo function (demo*.m).

- `demofun1` (demo solving `objfun1`, employs an EA for real valued parameters),
- `demofun2` (demo solving `objfun2`, employs an EA with ES as mutation operator),
- `demogeatbx` (simple menu driven demo to more than 15 objective functions and 3 different EA).

Inside a demo function all parameters of the EA and the optimization process can be defined. Every aspect of the behavior of the GEATbx can be controlled from the outside via parameters set in the demo functions. Thus, all things necessary to define a large optimization or to define special demonstrations are provided by the demo functions. This is the ideal point to define tailored parameter sets for public demonstrations, demos during lectures or to present special aspects of a large system.

5.4 Toolbox functions (Predefined algorithms)

The GEATbx contains a number of predefined EA. They are not more than a set of parameter options defined in a Toolbox function (tbx3*.m). The whole process of option setting, the meaning of the options and more is described in the document Parameter options of the GEATbx (3.x).

Examples of toolbox functions:

- `tbx3real` / `tbx3int` / `tbx3bin` (parameter for real / integer / binary valued variables),
- `tbx3es1` (implements an evolution strategy),
- `tbx3comp` (switches migration and competition on),
- `tbx3perm` (implements a simple permutation / ordering EA).

By using one of these functions the respective options are set at ones. This makes reading the demo functions easier. And the options set defined in the toolbox functions can be reused by other optimizations. Further parameters can be set in the demo functions, even the ones provided by the toolbox function overwritten. All options not defined are set automatically inside the main function `geamain2` by calling `geaoptset`, checking the validity of the options at the same time.

5.5 Evolutionary Algorithm - Main function

The demo functions call the working horse of the GEATbx:

- `geamain2` (GEA toolbox MAIN function)

Here all the administrative work is done: resolve parameter values, initialize the population, run the evolutionary algorithm including call of the objective function, and display and saving the results.

5.5.1 Initialization

The initialization is done by:

- `initpop` (high-level initialization function, initializes a whole population, includes user provided individuals, employs randomization of provided individuals and handles other aspects of initialization/inoculation),

which calls the representation specific low-level initialization functions:

- `initrp` (initializes a population using real values),
- `initip` (initializes a population using integer values),
- `initbp` (initializes a population using binary values).

An application specific initialization function (for instance `initdopi`) may also be employed. However, the already created/initialized individuals can be directly input to the main function (4th input parameter) via the demo function. The high-level initialization function `initpop` handles all aspects of population size, additional randomization or inclusion of randomly generated individuals.

5.5.2 Generational loop of the EA

As soon as the population of individuals is initialized and evaluated, the evolutionary algorithm starts. For as many generations as necessary/defined, new populations are produced and evalu-

ated. All functions of the evolutionary algorithm are called via high-level functions, thus supporting the multi-population (distributed) concept.

- Fitness assignment by ranking (`ranking`), see Subsection 5.5.3,
- Selection (`selection`), see Subsection 5.5.4,
- Recombination / Crossover (`recombin`), see Subsection 5.5.5,
- Mutation (`mutate`), see Subsection 5.5.6,
- Evaluation (obj*.m), see Subsection 5.5.7,
- Reinsertion (`reins`), see Subsection 5.5.8,
- Migration (`migrate`), see Subsection 5.5.9,
- Competition (`compete`), see Subsection 5.5.10,
- Visualization (`resplot`), see Subsection 5.5.11.

Inside the high-level functions the appropriate functions are called.

5.5.3 Fitness assignment by ranking

For fitness assignment by ranking the toolbox provides one function, no low-level functions are used:

- `ranking` (linear and non-linear ranking; single- and multi-objective ranking).

Low-level functions are only used for multi-objective ranking, especially GOAL attainment and sharing:

- `rankgoal` (perform goal preference calculation between multiple objective values),
- `rankplt` (RANK two multi objective values Partially Less Than),
- `rankshare` (SHARing between individuals).

5.5.4 Selection

For selection a high-level function is provided:

- `selection` (high-level selection function).

By parameter low-level selection functions can be chosen, which will be called for the subpopulations inside the high-level function:

- `selsus` (selection by stochastic universal sampling),
- `selrws` (roulette wheel selection),
- `seltrunc` (truncation selection),
- `seltour` (tournament selection),
- `sellocal` (local selection).

5.5.5 Recombination/Crossover

For recombination/crossover a high-level function is provided:

- `recombin` (high-level recombination/crossover function)

By parameter low-level recombination/crossover functions can be chosen which will be called for the subpopulations inside the high-level function. The term recombination is now used for all recombination functions (other publications use the term crossover to refer to recombination of binary valued variables).

Recombination for all parameter optimizations:

- `recdis` (discrete recombination).

Recombination for real valued parameter optimizations:

- `recint` (intermediate recombination),
- `reclin` (line recombination),
- `reclinex` (extended line recombination).

Recombination for binary valued parameter optimizations (all implemented in `recmp`, they are special cases of discrete recombination):

- `recsp` (single point crossover),
- `recdp` (double point crossover),
- `recsh` (shuffle point crossover),
- `recsprs` (single point crossover with reduced surrogate),
- `recdprs` (double point crossover with reduced surrogate),
- `recshrs` (shuffle point crossover with reduced surrogate).

Currently, for integer valued parameter optimizations `recdis` must be used.

Recombination for permutation/ordering/combinatorial optimizations:

- `recgp` (generalized position crossover/recombination),
- `recpm` (partial matching crossover/recombination).

5.5.6 Mutation

For mutation a high-level function is provided:

- `mutate` (high-level mutation function)

Depending on the variable representation one of the low-level mutation functions will be called for every subpopulation:

Real values:

- `mutreal` (mutation operator for real values)

Integer values:

- `mutint` (mutation operator for integer values)

Binary values:

- `mutbin` (mutation operator for binary values)

For real valued representation a number of additional mutation functions are provided. These functions use the mutation operators of evolution strategies. Thus, an adaptation of strategy parameters takes place.

Evolution strategy mutation operators:

- `mutes1` (adaptation of mutation step sizes),
- `mutes2` (adaptation of mutation step sizes and another set of strategy parameters).

Mutation for permutation/ordering/combinatorial optimizations (all implemented in `mutcomb`):

- `mutswap` (exchange/swap variables),
- `mutinvert` (invert parts of an individual),
- `mutmove` (move one variable inside an individual).

5.5.7 Evaluation

The toolbox provides many examples for objective functions. These functions are called "obj*.m". All functions use the same calling syntax.

Standard evolutionary algorithm test functions with a free definable dimension using the real valued representation are provided in:

- `objfun1` (DE JONG's function 1) - `objfun12` (MICHALEWICZ's function).

Binary valued representation is used in:

- `objone1` (ONEMAX function).

Dynamic optimization is implemented in:

- `objdopi` (double integrator),
- `objlinq2` (linear quadratic problem).

Standard optimization test functions with 2 independent variables (dimension = 2) are provided in:

- `objbran` (BRANIN's rcos function),
- `objeaso` (EASOM's function),
- `objgold` (GOLDSTEIN-PRICE function),
- `objsixh` (six hump camelback function).

A number of multi-objective functions are also provided:

- `mobjfonseca1` (FONSECA's MO test function 1),
- `mobjfonseca2` (FONSECA's MO test function 2),
- `mobjbelegundu` (BELEGUNDU's function (constrained)),
- `mobjdtlz1` (Deb, Thiele, Zitzler, Laumanns' multiobjective function 1),
- `mobjdtlz2` (Deb, Thiele, Zitzler, Laumanns' multiobjective function 2),
- `mobjdtlz3` (Deb, Thiele, Zitzler, Laumanns' multiobjective function 3),
- `mobjkita` (KITA's function (constrained)),
- `mobjquagliarella` (QUAGLIARELLA's function),
- `mobjcantilever` (multiobjective cantilever beam system),
- `mobjdebconstr` (DEB's constrained function),
- `mobjsoland` (SOLAND function multi-objective version).

A rudimentary interface to the tsp-lib is also provided (not fully implemented, take it as a starting point and send me your extended implementation):

- `objtsplib` (interface to the tsp-lib, see subdirectory objfun/tsp).

See *M-FUNCTIONS - INDEX* (part of online documentation) for all functions or *EXAMPLES OF OBJECTIVE FUNCTIONS* (part of online documentation) for a mathematical description.

It is possible to pass any number of additional parameters to the objective function, as long as the respective objective function can handle them. These parameters can be defined or loaded in the demo function and send as 5th parameter (and more) to `geamain2`. All these parameters are directly passed to the objective functions. This can be used to optimize a special aspect of the objective function or to define one or multiple parameter sets inside the objective function.

Instead of writing an objective function file and passing the name of the function, the objective function can be passed directly in a string to the main function as well. However, it is recommended to use one of the provided objective functions as a template and pass over only the file name.

5.5.8 Reinsertion

For reinsertion the toolbox provides a high-level function:

- `reins` (reinsertion of offspring into population).

Depending on the used population model (global/regional or local), a corresponding low-level reinsertion function is called from inside the high-level reinsertion function.

Low-level reinsertion functions:

- `reinsreg` (global/regional reinsertion of offspring into population),
- `reinsloc` (local reinsertion of offspring into population).

5.5.9 Migration

For migration the toolbox provides 1 function, no low-level functions are used:

- `migrate` (migration of individuals between subpopulations).

5.5.10 Competition

For competition between subpopulations the toolbox provides 1 function, no low-level functions are used:

- `compete` (competition between subpopulations).

5.5.11 Visualization

The visualization of the results is implemented on 3 different levels:

- tabular output of results of evolutionary algorithm (`geaoptprint` - pretty printed output of the options; `gearunstatus` - pretty printed output of the results of one generation),
- graphical output of results of evolutionary algorithm (`resplot`),
- problem specific output (specific m-file, for instance `plotdopi` for `objdopi`).

5.6 Utility functions

Throughout the toolbox low-level utility functions are used. They are useful for everyday work as well:

- `compdiv` (compute diverse things),
- `comploc` (compute stuff for the local model),
- `compplot` (compute stuff for plotting and visualization),
- `chkbound` (check boundaries and reset variables outside the boundaries),
- `savebindata` (save matlab variables using a provided name into one mat file),
- `plotstd` (set standard options for a new figure, used in all plot functions)
- `expandm` (expand a matrix),
- `prprintf` (pretty printed output),
- `straddname`/`straddtime` (add a name/string or the current time to a string or filename),
- `deblankall` (deblank both sides of a string),
- `findfiles` (find files inside multiple directory and file search masks including wildcard),
- `menutext` (same as menu, but without graphical stuff, thus, can be compiled).

For the handling of the options structure a few special functions are provided:

- `geaoptset` (set a GEAOpt option),
- `geaoptsave` (save a GEAOpt structure into a m-file or a text file),
- `geaoptload` (load GEA Options from a text file into a GEAOpt structure).

6 Data Structures of the GEATbx

Nearly all data structures of the GEATbx are mapped to 2-D matrizes.

The used data structures in the GEATbx:

- Chromosomes (genotype / individuals)
- Phenotype (decision variables / individuals)
- Objective function values (objective values)
- Fitness values

Remark:

Many problems don't need a mapping from the chromosome to phenotype structure. For instance, if the variables are real valued and the evolutionary algorithm works with this real valued variables, there is no mapping necessary. Similar for binary variables and an evolutionary algorithm that uses binary variables. Then, chromosomes and phenotypes are identical. Thus, throughout the whole documentation the term individual is used for both, chromosomes and phenotypes. Only if differentiation is necessary between both, the original terms will be used. Please read the section about Variable Representation, Chapter 4, p. for more information as well.

6.1 Chromosomes (genotype / individuals)

The chromosome data structure stores an entire population in a single matrix of size `Nind x Lind`, where `Nind` is the number of individuals in the population and `Lind` is the length of the genotypic representation of those individuals (for integer and real valued representation Lind is 1, that means, genotype and phenotype are identical). Each row corresponds to an individual's genotype, consisting of binary, integer or real values.

An example of the chromosome structure

$$
\texttt{Chrom} = \begin{matrix}
g_{1,1} & g_{1,2} & g_{1,3} & \ldots & g_{1,\texttt{Lind}} & \texttt{individual 1} \\
g_{2,1} & g_{2,2} & g_{2,3} & \ldots & g_{2,\texttt{Lind}} & \texttt{individual 2} \\
g_{3,1} & g_{3,2} & g_{3,3} & \ldots & g_{3,\texttt{Lind}} & \texttt{individual 3} \\
\vdots & \vdots & \vdots & \ldots & \vdots & \\
g_{\texttt{Nind},1} & g_{\texttt{Nind},2} & g_{\texttt{Nind},3} & \ldots & g_{\texttt{Nind},\texttt{Lind}} & \texttt{individual Nind}
\end{matrix}
$$

This data representation does not force a structure on the chromosome structure, only requiring that all chromosomes are of equal length. Thus, structured populations or populations with varying genotypic bases may be used in the GEATbx. However, a suitable decoding function, mapping chromosomes onto phenotypes, must be employed.

6.2 Phenotypes (decision variables / individuals)

The decision variables (phenotypes) in the evolutionary algorithm are obtained by applying some mapping from the chromosome representation into the decision variable space. Here, each string

contained in the chromosome structure decodes to a row vector of order `Nvar`, according to the number of dimensions in the search space and corresponding to the decision variable vector value.

The decision variables are stored in a numerical matrix of size `Nind x Nvar`. Again, each row corresponds to a particular individual's phenotype. An example of the phenotype data structure is given below, where `bin2real` is used to represent a decoding function mapping the genotypes onto the phenotypes.

```
Phen = bin2real(Chrom)    % map genotype to phenotype
Phen = x1,1      x1,2      x1,3     ...  x1,Nvar        individual 1
       x2,1      x2,2      x2,3     ...  x2,Nvar        individual 2
       x3,1      x3,2      x3,3     ...  x3,Nvar        individual 3
        .         .         .       ...   .
       xNind,1   xNind,2   xNind,3   ...  xNind,Nvar     individual Nind
```

The actual mapping between the chromosome representation and their phenotypic values depends upon the `decode` function used. It is perfectly feasible using this representation to have vectors of decision variables of different types. For example, it is possible to mix integer, real-valued, and binary decision variables in the same `Phen` data structure.

6.3 Objective function values

An objective function is used to evaluate the performance of the phenotypes in the problem domain. Objective function values can be scalar or, in the case of multiobjective problems, vectors. Note that objective function values are not necessarily the same as fitness values.

Objective function values are stored in a numerical matrix of size `Nind·Nobj`, where `Nobj` is the number of objectives. Each row corresponds to a particular individual's objective vector. An example of the objective function values data structure is shown below, with `mobjfonseca2` representing an example multiobjective function.

```
ObjV = mobjfonseca2(Phen)     % objective function
ObjV = y1,1      y1,2      y1,3     ...  y1,Nobj        individual 1
       y2,1      y2,2      y2,3     ...  y2,Nobj        individual 2
       y3,1      y3,2      y3,3     ...  y3,Nobj        individual 3
        .         .         .       ...   .
       yNind,1   yNind,2   yNind,3   ...  yNind,Nobj     individual Nind
```

6.4 Fitness values

Fitness values are derived from the objective function values through a scaling or ranking function. Fitness values are non-negative scalars and are stored in column vectors of length `Nind`, an example of which is shown below. `ranking` is a fitness function contained in the GEATbx.

```
Fitn = ranking(ObjV)     % fitness function
Fitn = f1      individual 1
       f2      individual 2
       f3      individual 3
        ...
       fNind   individual Nind
```

Note that for multiobjective functions, the fitness of a particular individual is a function of a vector of objective function values. Multiobjective problems are characterized by having no single unique solution, but a family of equally fit solutions with different values of decision variables. Care should therefore be taken to adopt some mechanism to ensure that the population is able to evolve the set of Pareto optimal solutions.

6.5 Multiple subpopulations

The GEATbx supports the use of a single population divided into a number of subpopulations or demes by modifying the use of data structures so that subpopulations are stored in contiguous blocks within a single matrix. For example, the chromosome data structure, `Chrom`, composed of `Subpop` subpopulations, each of length `N` individuals `Ind`, is stored as:

$$
\texttt{Chrom} = \begin{matrix}
\cdots \\
\texttt{Ind}_1\ \texttt{Subpop}_1 \\
\texttt{Ind}_2\ \texttt{Subpop}_1 \\
\cdots \\
\texttt{Ind}_N\ \texttt{Subpop}_1 \\
\texttt{Ind}_1\ \texttt{Subpop}_2 \\
\texttt{Ind}_2\ \texttt{Subpop}_2 \\
\cdots \\
\texttt{Ind}_N\ \texttt{Subpop}_2 \\
\cdots \\
\texttt{Ind}_1\ \texttt{Subpop}_{\texttt{Subpop}} \\
\texttt{Ind}_2\ \texttt{Subpop}_{\texttt{Subpop}} \\
\cdots \\
\texttt{Ind}_N\ \texttt{Subpop}_{\texttt{Subpop}}
\end{matrix}
$$

This is known as the *regional model*, also called *migration* or *island model*.

7 How to Approach new Optimization Problems

You are faced with a new optimization problem. How are you best going to approach this task? What questions have to be asked? What do you have to consider? What aspects do you have to examine?

In this section I attempt to provide step-by-step instructions for solving optimization problems. Every single step is illustrated with examples, and you will find cross-references to other sections offering further examples.

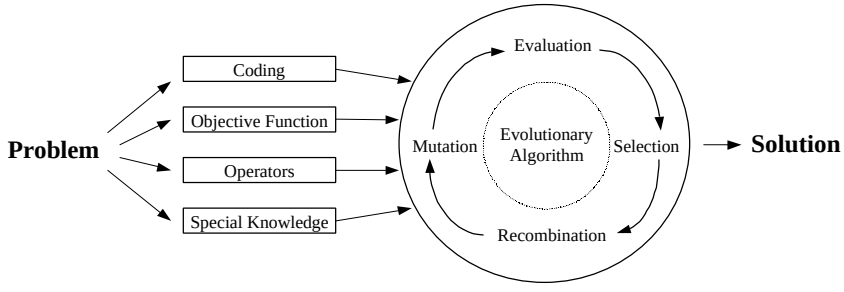
The approach I am presenting here is based on my own experiences gained in the course of solving real-world optimization problems using the GEATbx. I have received lots of positive feedback from many others who found this approach helpful for solving complex problems.

The main points are:

- classifying the problem and defining the objective function,
- preliminary investigation of the system behavior,
- selecting the appropriate optimization approach based on preliminary investigations (Chapter 'Combination of Operators and Options to Produce Evolutionary Algorithms' in "GEATbx Introduction" discusses this point more closely),
- executing and evaluating optimizations.

Fig. 7-1 illustrates these points. The first thing to do when confronted with a new problem is to analyze it. This will provide you with the coding, number and boundaries of variables. Now you can implement the objective function. Very often you will have to gather further information on the system behavior. Once you have a basic understanding of the system you can start selecting operators and parameters of the evolutionary algorithm. With this information you have the foundation to start executing the optimization.

Fig. 7-1. Procedure for solving optimization problems using evolutionary algorithms



Some of these steps can be utilized with other optimization tools or methods as well. Additionally, I have included several alternatives to investigating the system behavior. These help gain further information about the problem and can also be applied without optimization. These tools are mostly included in the GEATbx.

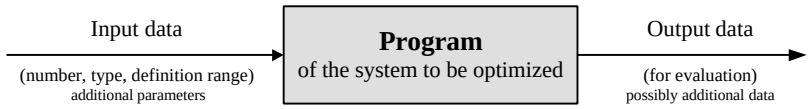
7.1 Classifying the Problem and Defining the Objective Function

When you begin to work on a new application you first need to classify the optimization problem. Consider how the system you want to investigate is defined and which outbound interfaces exist or have to be defined. This means you have to determine:

- input data, including type and domain,
- additional parameters necessary for system behavior control, which are not changed during the optimization,
- output data necessary for the evaluation of the system.

The system has to be available as a program which is entirely controlled through the input data (and possibly the additional parameters) and which as the result of the execution provides the output data. This program is the implementation of the objective function. Fig. 7-2 represents this structure graphically.

Fig. 7-2. Structure of the system to be optimized as objective function



The objective function is not only applicable within the optimization. It can also be applied during program/system simulation with input data, e.g.:

- interactive testing using systematically determined input data, e.g. call of the objective function with a data record and output of the objective value,
- automatic simulation of several data records, e.g. used for direct visualization of the objective function, as well as Section 7.2, p.),
- problem-specific visualization, for instance simulation of the objective function and evaluation of possible additional output parameters (e.g. states of a dynamic system) providing additional information on the system.

It is very important that the objective function is implemented to be universally applicable to all these areas of application! You will find that using just one program for the problem at hand (optimization, problem-specific visualization, single simulation) pays off and also facilitates program maintenance.

7.2 Investigating the System Behavior

The basic questions when beginning to work on a new problem are: what are the system properties or, for that matter, the objective function properties, what does the search space look like, and do specific features occur. Often these questions only crop up once the first optimization has been carried out and the hoped for results have not been obtained.

The results of the investigations suggested in the following could offer new information on the system behavior as well as indicate the type of objective function and the optimization procedure needed for a successful optimization.

The system at hand generally poses a high-dimensional problem for the optimization. An objective function value, subject to the given parameters, is returned as result of the simulation and application of the objective function.

The generally applicable investigations of the system behavior can be subdivided as follows:

1. one and two-dimensional slices through the objective function,
2. multi-dimensional visualization of the objective function,
3. possible decrease of system size/dimension.

I will take a closer look at these areas in the following.

One and Two-dimensional Slices (Variational Diagrams)

The systems under test are mostly high-dimensional (more than five dimensions or five parameters). That is why a direct graphic representation of the search space is not possible. However, it is possible to illustrate some features of the objective function by creating one or two-dimensional slices. The results of slicing can be visualized using standard diagrams.

The slices through the objective function are made through a certain point in the search space (here called center point). To calculate the slices through the center point all non-varying parameters of the system are kept constant to the values of the center point. The varying or free parameters are changed within an area around the center point. The boundaries of this area determine the extent of the slices as well as the level of detail. The area around the center point is discretized, the corresponding objective function values calculated, and the results visualized in 2-D and 3-D graphics.

The center point can be the result of an optimization and thus represent a very good point, or it can be a point of great importance within the search domain.

As the standard visualization methods are limited to three dimensions, only one-dimensional (variation of one parameter) and two-dimensional slices (variation of two parameters) can be carried out. One-dimensional slices are easy to calculate. But they can only offer information along the coordinate axis of this one varied parameter. Two-dimensional slices need clearly more calculations, however, they offer additional information about the interaction/correlation between the two varied parameters.

The following information can be deduced from the slices through the objective function:

- an outline of the objective function,
- whether the objective function is rough or smooth,
- whether there is more than one minimum in the objective function (existence of local minima),
- whether there is a correlation between single variables,
- which variables are very sensitive, and which ones have hardly any influence on the objective function,
- restricting the area of interest (important for new problems where the boundaries of the variables are little or not known).

However, keep in mind that all information drawn from these slices is only valid for the represented area and subject to restrictions. Due to the oftentimes non-linear relations between parameters it is possible to deduce different information for different areas. The gathered information is

thus only valid for the examined areas. The results may be valid for other areas too, however this need not be the case.

For the parameter optimization of different real-world systems, variational diagrams were used for investigating the system behavior. Currently they are only explained in detail in Chapter 8 of [Poh99b] (greenhouse control system, car steering example).

Multi-dimensional Visualization

Most of the commonly used techniques for visualization are limited to representing data depending on one or two variables. This is due to the human visual limitation to three dimensions. There are two possible extensions to go beyond this limitation: using color for the fourth dimension and time as the fifth dimension. Neither possibility is very common and requires practice, especially if time is used for visualizing the fifth dimension. However, if the problem incorporates more than five dimensions a new method for visualizing arbitrarily high dimensions must be found.

For the visualization of multi-dimensional data a method to transform multi-dimensional data to a lower dimension is needed, preferably to 2 or 3 dimensions. This transformation should provide a lower-dimensional picture where the dissimilarities between the data points of the multi-dimensional domain corresponds with the dissimilarities of the lower-dimensional domain.

These transformation methods are referred to as *multi-dimensional scaling*. One well-known multi-dimensional scaling method is the SAMMON-Mapping method [Sam69].

Please note that all these procedures are subject to information loss. On the one hand, visualization of the information is only possible in this way. On the other hand, we have to take the reduction of information into consideration when interpreting the results.

The multi-dimensional visualization can be applied to the following tasks:

- representation of the similarity of results in the variable domain and the corresponding objective function values (comparing multiple optimization results),
- representation of the “way of the improving solution through the variable search space” („Weg der sich verbessernden Lösung durch den Variablen-Suchraum“) during an optimization,
- representation of the “way of the improving solution through the objective function domain” („Weges der sich verbessernden Lösung durch den Zielfunktions-Suchraum“) during optimization (for multi-criteria problems),
- comparison of the „ways through the search space“ of different optimizations,
- attempt to generate a low-dimensional representation of the search space of the objective function.

The first four tasks can be calculated and subsequently visualized with the aid of functions which are included in the GEATbx. Currently, the last variant still comes across calculation and visualization problems. Even for an objective function with only ten dimensions or variables large data amounts have to be processed.

These methods enable a clear illustration of complex data relations where other methods either fail or require a great deal of time and familiarization.

Decreasing the System Size/Dimension

An important aspect of investigating new systems is the analysis of the size/dimensionality of the problem and the possible means of its reduction. Most real-world systems are high-dimensional. Often, it is possible to reduce the number of dimensions for the optimization without distorting the results. If this is not directly possible try and perform first optimizations with a reduced or limited system to obtain a better impression of the overall system behavior.

Which variants of dimension reduction or difficulties should you examine?

- Reduction of the number of variables:
  Some of the existing variables are omitted (reduction of problem dimensions) or specified as set values (reduction of the size of the optimization problem). The reduction of the number of variables is the most important variant which should be taken full advantage of!
- Scalability of the problem:
  System variables often represent discrete values of a course to be optimized. Depending on the resolution of the discretization the number of the variables of the optimization problem will change. I recommend a rough discretization for the first optimizations which can be adjusted to the real requirements later.
  On the other hand, for some problems it is also possible to start off with a reduced number of variables without distorting the nature of the problem.
- Subdividing the problem into several optimization tasks:
  Dividing an optimization task into several ones offers great advantages. In this way the overall problem can be split up into several more manageable tasks, an option that is definitely of advantage. The overall temporal horizon is divided into several sections. The final state of one section represents the initial state of the next section. This method enables us to solve problems which cannot be mastered with current methods and devices. However, this approach is problem-specific. Keep in mind that the division into sections might effect the result of the optimization.

A few examples of the GEATbx apply the above mentioned methods. Currently they are only explained in detail in Chapter 8 of [Poh99b] (greenhouse control system, car steering example).

7.3 Selecting the Optimization Method

With our knowledge of the type of the system at hand and the findings of the preliminary examination on the system behavior (Section. 7.2, p.) we now have to choose an optimization method. This can be a single method, or a combination of several methods, or variants of one or more methods.

The number of different problem classes is huge, and for each problem class a different method might be most suitable.

Chapter 'Combination of Operators and Options to Produce Evolutionary Algorithms' in "GEATbx Introduction" deals with the selection procedure of methods and operators as well as the specification of appropriate options/parameters. Section 'Generally Adjustable Operators and Options' in "GEATbx Introduction" gives you indications on methods and operators which can be used for most problem classes. Section 'Globally Oriented Parameter Optimization' in "GEATbx Introduction" onwards deals with the definition of optimization methods for special problem

classes. All suggestions are based on my own experiences and my work on different systems as well as on results found in the literature.

7.4 Executing and Evaluating Optimizations

Once you have selected one or more optimization methods, first optimizations can be carried out. These can already provide information on how easily the problem can be solved.

Should the first trials already provide the optimal solution (with knowledge of the optimum) or a good or sufficient result, we can consider the problem as solved. Of course, depending on further requirements it is also possible to apply the methods described in the following to find the solutions even more easily.

More often than not, a problem cannot be considered solved after the first runs. Either the results are unsatisfactory or, if the optimum is not known, there is no way of being certain about the obtained results. This is where you should start to find methods which are better suited.

Basically, there are two approaches: either different methods, operators or options are used, or the number of individuals and/or sub-populations is increased. The latter procedure leads to a clear increase in calculation time because the number of individuals linearly effects the overall calculation time (at least in most real problems). If the total calculation time is still less than 24 hours it at least resolves the question as to what extent it is possible to find a solution in this way.

Should the methods used so far prove unsuccessful new variants have to be found. At this point it is necessary to reconsider the system properties. If the system cannot be optimized the way we had hoped for the system must have properties which have not been included in the selection process. By applying further methods it is possible to obtain additional information about the problem at hand.

The visualization possibilities can be an important help when evaluating executed runs. These permit a detailed insight into the state of the population during various stages in an optimization as well as the course of the optimization. Questions as to an early convergence of the population, the existence of several extreme values, or the curve of variables of good individuals find answers here. You need to familiarize yourself with the application of these methods in order to be able to interpret the diagrams. The visualization methods offer the best possible insight into the current state and course of the optimization process and open up new means to adapt the parameters used in the optimization procedure. You can find examples of their application in the following Sections.

At this point I would like to refer to the simultaneous application of different strategies during an optimization run. Please see Chapter 'Application of different strategies' in "GEATbx Introduction" for more details. This procedure on the one hand enables you to apply more than one strategy to solve a problem. On the other hand, it is often the case that, in dependence of the progress of the optimization, different methods are better suited for different stages within the optimization run. The strategies thus support each other, i.e. the success of one strategy enables the subsequent success of another one.

One good example to illustrate this is the application of several mutation variants, where one variant performs a rough search, another one a more refined search, and yet another one an even more refined search. A similar process can be achieved by the simultaneous use of different re-

combination operators. During the subsequent evaluation it can very easily be checked at what time which of the applied strategies was successful. These means enable us to find the most promising methods for solving a problem. These can in turn be further refined to solve the problem at hand in the best possible and least time-consuming way.

If you find that after having applied the existing and known methods the results are still unsatisfactory you have to consider including further problem-specific knowledge. For example:

- the special initialization of the individuals of the initial population (Section 5.5.1, p., more info tbd.),
- the adapted restriction of value domains of the variables, and
- the development of special operators.

The first two points should always be applied, in case this knowledge is available. This usually leads to a clear reduction of the search domain and consequently to an acceleration of finding satisfactory results. The development of special operators is usually very difficult and has, as far as current problems go, not been necessary.

The methods I have introduced for solving optimization problems and the represented variables should make it possible to solve many occurring engineering problems. You need more time to become familiar with bigger and more complex problems. The problem has to be examined closely during different optimization trials. Once the methods and means have been found to solve the problem we have often also obtained new knowledge on the system which was not known even to the experts before the optimization. The optimization process examines such areas of the system which were not taken into consideration beforehand. This increase in knowledge should not be underestimated, neither by the experts of the system to be optimized, nor the system engineer who executes the optimization.

If, after having occupied yourself extensively with the system, you find that the method of evolutionary algorithms seems most appropriate, use it. However, if a problem can be solved with a special method (e.g. gradient based methods) this means that a solution will usually be found more quickly than by using evolutionary algorithms. However, these special methods often require system properties which are hardly guaranteed in real systems (rectangular shapes, differentiability, etc.). In these cases, evolutionary algorithms offer a large spectrum of efficient strategies and operators for solving the optimization problem.