NGPM -- A NSGA-II Program in Matlab

Version 1.4

LIN Song

Aerospace Structural Dynamics Research Laboratory

College of Astronautics, Northwestern Polytechnical University, China

Email: lsssswc@163.com

2011-07-26

Contents

1. Introduction

This document gives a brief description about NGPM. NGPM is the abbreviation of "A NSGA-II Program in Matlab", which is the implementation of NSGA-II in Matlab. NSGA-II is a multi-objective genetic algorithm developed by K. Deb[1] . The details of NSGA-II are not described in this document; please refer to [1] . From version 1.3, R-NSGA-II — a modified procedure of NSGA-II — is implemented, the details of R-NSGA-II please refer to [2] .

2. How to run the code?

To use this program to solve a function optimization problem. Optimization model such as number of design variables, number of objectives, number of constraints, should be specified in the NSGA-II optimization options structure[1] which is created by function `nsgaopt()`. The objective function must be created as a function file (*.m), and specify the function handle *options.objfun* to this function. The Matlab file `TP_CONSTR.m` is a script file which solves a constrained test function. This test problem is 'CONSTR' in [1] .

2.1. 'CONSTR' test problem description

(1) Objectives: 2

$$\begin{aligned} f_1(x) &= x_1 \\ f_2(x) &= (1+x_2)/x_1 \end{aligned} \tag{1}$$

(2) Design variables: 2

$$x_1 \in [0.1,1.0],\ x_2 \in [0,5] \tag{2}$$

(3) Constraints: 2

$$\begin{aligned} g_1(x) &= x_2 + 9x_1 \geq 6 \\ g_2(x) &= -x_2 + 9x_1 \geq 1 \end{aligned} \tag{3}$$

Two steps should be done to solve this problem.

2.2. Step1: Specified optimization model

The file `TP_CONSTR.m` is a script file which specified the optimization model.

[1] In this document, all of the italic *options* is the structure created by `nsgaopt()`.

```
% TP_CONSTR.m file
% 'CONSTR' test problem
clc; clear; close all

options = nsgaopt();                    % create default options structure
options.popsize = 50;                   % population size
options.maxGen = 100;                   % max generation

options.numObj = 2;                      % number of objectives
options.numVar = 2;                      % number of design variables
options.numCons = 2;                     % number of constraints
options.lb = [0.1   0];                  % lower bound of x
options.ub = [1      5];                 % upper bound of x
options.objfun = @objfun;                % objective function handle

result = nsga2(options);                % start the optimization progress
```

2.3. Step2: Create a objective function

The file `TP_CONSTR_objfun.m` is a function file which specified the objective function evaluation. The objective function is specified by *options.objfun* parameter created by the function `nsgaopt()`. Its prototype is:

```
[y, cons] = objfun(x, varvargin)
```

x : Design variables vector, its length must equals *options.numVar.*

y : Objective values vector, its length must equals *options.numObj*.

cons : Constraint violations vector. Its length must equals *options.numCons*. If there is no constraint, return empty vector [].

varargin : Any variable(s) which are passed to `nsga2` function will be finally passed to this objective function. For example, if you call

```
result = nsga2(opt, model, param)
```

The two addition parameter passed to `nsga2` — *model* and *param* — will be passed to the objective function as

```
[y, const]=objfun(x, model, param)
```

In this function optimization problem, there is no other parameter.

```
function [y, cons] = objfun(x)
% TP_CONSTR_objfun.m file
% 'CONSTR' test problem
y = [0,0];
cons = [0,0];

y(1) = x(1);
y(2) = (1+x(2)) / x(1);

% calculate the constraint violations
c = x(2) + 9*x(1) - 6;
if(c<0)
    cons(1) = abs(c);
end

c = -x(2) + 9*x(1) - 1;
if(c<0)
    cons(2) = abs(c);
end
```

2.4. Results

Run the script file `TP_CONSTR.m`, and you will get the optimization result store in the *result* structure. The population will be plotted in a GUI figure. The x-axis is the first objective, and the y-axis is the second. If user specifies the names of objectives in *options.nameObj*, then they will be displayed in the x and y labels.

On the GUI window 'plotnsga', the optimization progress could be paused or stop by press the corresponding buttons. Note that, the progress of optimization would pause or stop only when the current population evaluation is done!

The Pareto front (or population distribution) of generation 1 and 100 was plot in Fig. 1. The populations of each generation were outputted to the file 'populations.txt' in current path.

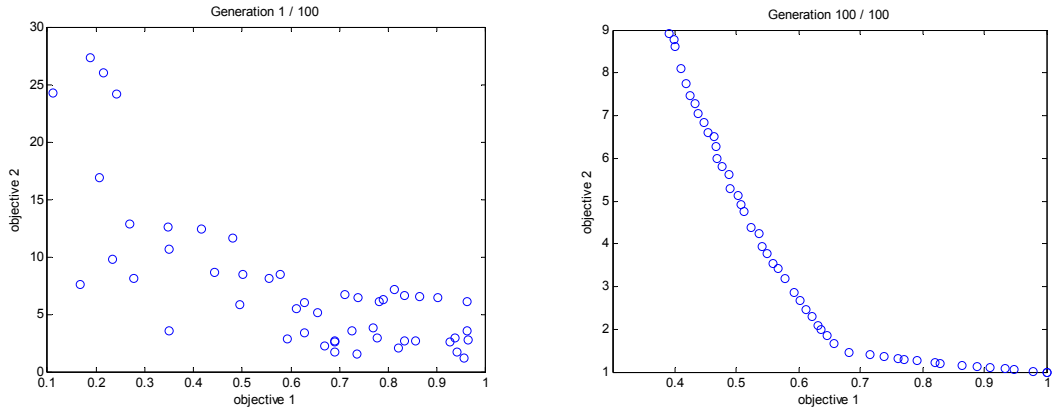


Fig. 1: Optimization results

In the population file, there is a head section in the beginning which saves some information of optimization model. The head section begins with "#NSGA2" line, and ends with "#end" line. And there is a state section in the front of each generation of population, which begins with "#Generation" line and ends with "#end" line.

Populations.txt file example:

```
#NSGA2
popsize 50
maxGen 100
numVar 2
numObj 2
numCons 2
stateFieldNames currentGen  evaluateCount   totalTime    firstFrontCount
    frontCount   avgEvalTime
#end



#Generation 1 / 100
currentGen  1
evaluateCount   50
totalTime    0.534723
firstFrontCount   7
frontCount   22
avgEvalTime      5.4181e-005
#end
```

```
Var1     Var2     Obj1     Obj2     Cons1   Cons2
0.833251    4.528960.833251    6.6354  0    0
0.214288    4.566880.214288    25.97850    3.63829
0.669123    0.487702    0.669123    2.223360    0
0.350648    2.734410.350648    10.65    0.109757    0.578572
0.961756    4.824440.961756    6.056050    0
0.241852    4.852960.241852    24.20060    3.6763
...
```

3. NGPM Options

This program is written for finite element optimization problem, the "Intermediate crossover" and "Gaussian mutation" is adequate for my use. Thus, I don't implement other genetic operators into NGPM. The real/integer coding, the binary tournament selection, the Gaussian mutation operator and the intermediate crossover operator work well in my application. If you want to use other genetic operators, try to modify the code yourself.

The following genetic operators and capabilities are supported in NGPM:

3.1. Coding

Real and integer coding are both supported. If the coding types of design variables are not specified in *options.vartype* vector, real coding is use as default.

options.vartype: integer vector, the length must equal to the number of design variables. 1=real, 2=integer. For example, [1 1 2] represents that the first two variables are real, and the third variable is integer.

3.2. Population options

options.popsize : even integer, population size.

options.numVar : integer, number of design variables.

options.numOb : integer, number of objectives.

options.numCons : integer, number of constraints.

options.lb : vector, lower bound of design variables, the length must equal to *numVar*.

options.ub : vector, upper bound of design variables , the length must equal to *numVar*.

3.3. Population initialization

There are three ways to initialize the population:

(1) **(default)** Using uniform distribution random number between the lower and upper bounds.

(2) Using population file generated in previous optimization.

(3) Using the population result structure in previous optimization.

All these approach are specified in the *options.initfun* cell array parameter.

3.3.1. Uniform initialization

```
options.initfun = {@initpop}
```

Description:

Create a random initial population with a uniform distribution. This is the default approach.

3.3.2. From exist population file

```
options.initfun={@initpop, strFileName, ngen}
```

strFileName : string, the optimization result file name.

ngen : (optional) integer, the generation of population would be used. If this parameter is not specified, the last population would be used.

Description:

Load population from exist population file and use the last population. If the popsize of the population from file less than the popsize of current optimization model, then uniform initialization would be used to fill the whole population.

Example:

```
options.initfun={@initpop, 'pops.txt'}          % Restart from the last generation
options.initfun={@initpop, 'pops.txt', 100}     % Restart from the 100 generation
```

3.3.3. From exist optimization result

```
options.initfun={@initpop, oldresult, ngen}
```

oldresult : structure, the optimization result structure in the workspace.

ngen : (optional) integer, the generation of population would be used. If this parameter is not specified, the last population would be used.

Description: Load population from previous optimization result structure. The result structure can be:

1. The result generated by last optimization procedure.
2. The result loaded from file by `loadpopfile('pop.txt')` function.
3. The *oldresult* generated in the global workspace by the `plotnsga('pop.txt')` function. (The `plotnsga` function calls `loadpopfile` function too.)

Example:

```
oldresult=loadpopfile('pop.txt');
options.initfun={@initpop, oldresult}          % Restart from the last generation
options.initfun={@initpop, oldresult, 100}     % Restart from the 100 generation
```

3.4. Selection

Only binary tournament selection is supported.

3.5. Crossover

Only intermediate crossover[3] (which also names arithmetic crossover) is supported in the current version.

3.5.1. Crossover fraction

options.crossoverFraction: scalar or string, crossover fraction of variables of an individual. If 'auto' string is specified, NGPM would used 2/numVar as the *crossoverFraction*.

NOTE: All of the individuals in the population would be processed by crossover operator, and only *crossoverFraction* of all variables would do crossover.

3.5.2. Intermediate crossover

```
options.crossover={'intermediate', ratio};
```

Intermediate crossover [3] creates two children from two parents: parent1 and parent2.

child1 = parent1 + rand×ratio×(parent2 - parent1)

child2 = parent2 - rand×ratio× (parent2 - parent1)

ratio: scalar. If it lies in the range [0, 1], the children created are within the two parent. If algorithm is premature, try to set *ratio* larger than 1.0.

3.6. Mutation

Only Gaussian mutation (which also names normal mutation) is supported in the current version.

3.6.1. Mutation fraction

options.mutaionFraction: scalar or string, mutation fraction of variables of an individual. If 'auto' string is specified, NGPM would use 2/numVar as the *mutaionFraction*.

NOTE: It's similar to the *crossoverFraction* parameter described before. All of the individuals in the population would be processed, and only *mutaionFraction* of all variables would do mutation.

3.6.2. Gaussian mutation

```
options.mutation = {'gaussian', scale, shrink}
```

Gaussian mutation[3] adds a normally distributed random number to each variable:

$$child = parent + S \times randn \times (ub - lb);$$

$$S = scale \times (1 - shrink \times currGen / maxGen);$$

scale: scalar, the *scale* parameter determines the standard deviation of the random number generated.

shrink: scalar, [0,1]. As the optimization progress goes forward, decrease the mutation range (for example, *shrink*$\in$[0.5, 1.0]) is usually used for local search. If the optimization problem has many different local Pareto-optimal fronts, such as ZDT4 problem[1], a large mutation range is require getting out of the local Pareto-optimal fronts. It means a zero shrink should be used.

3.7. Constraint handling

NGPM uses binary tournament selection based on constraint-dominate definition to handle constraint which proposed by Deb[1].

3.8. Stopping Criteria

Only maximum generation specified by *options.maxGen* is supported currently.

Example:

```
options.maxGen=500;
```

3.9. Output function

3.9.1. Output function

In current version NGPM, the only output function is `output2file` which outputs the whole population includes design variables, objectives and constraint violations (if exist) into the specified file (*options.outputfile*).

options.outputfuns: cell array, the first element must be the output function handle, such as *@output2file*. The other parameter will be passed to this function as variable length input argument. The output function has the prototype:

```
function output (opt, state, pop, type, varargin)
```

opt : the *options* structure.

state : the *state* structure.

pop : the current population.

type : use to identify if this call is the last call.

-1 = the last call, use for closing the opened file or other operations.

others(or no exist) = normal output

varargin : the parameter specified in the *options.outputfuns* cell array. There is no parameter for the default output function `output2file`.

3.9.2. Output interval

options.outputInterval : integer, interval between two calls of output function. This parameter can be assigned a large value for efficiency.

Example:

```
options.outputInterval = 10;
```

3.10. GUI control

The GUI window 'plognsga' (Fig. 2) is use to plot the result populations or control the optimization progress. Call `plotnsga` function to plot the populations:

```
plotnsga(result)
plotnsga(strPopFile)
```

result : structure created by `nsga2`() function.

strPopFile : population file name.

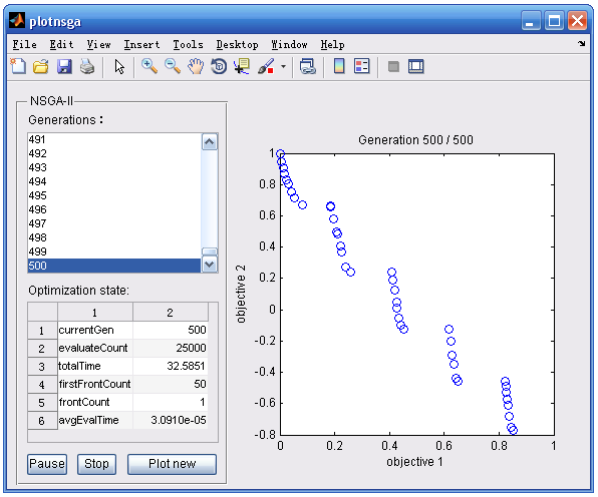


Fig. 2: plotnsga GUI window

3.10.1. Pause, stop

Press “Pause” button to pause the optimization progress, and the button title would be changed to “Continue”. Then, press “Continue” button to continue the optimization.

Press “Stop” button to stop the optimization progress. This operation would stop the nsga2 iteration and closed the output file if specified.

NOTE: When ‘pause’ or ‘stop’ button is pressed, the program would response until the current generation of optimization progress is finished.

3.10.2. Plot in new window

Press “Plot new” button to plot the selected population in a new figure window. This function is designed to save the figure as EMF file (because the window could not be saved as EMF file if there is any GUI control on the figure window).

3.10.3. Optimization state

The optimization state list-box lists all fields of the *state* structure of selected generation.

Table 1: Optimization states

Name	Description
currentGen	The selected generation ID, begin with 1.
evaluateCount	The objective function evaluation count from the beginning.
totalTime	The elapsed time from optimization progress start.
firstFrontCount	The number of individuals in the first front.
frontCount	The number of fronts in current population.
avgEvalTime	The average evaluation time of current generation.

3.10.4. Load from result

```
plotnsga(strPopFile)
```

strPopFile : string, the optimization result file name.

Description:

The function `plotnsga` will first call `loadpopfile`() function to read the specified optimization result file. A global variable named "*oldresult*" which contains the optimization result in the file would be created in global workspace. Then the population loaded from file would be plotted in the GUI window, and the file name was showed in the figure title.

Example:

```
plotnsga('populations.txt')
```

3.11. Plot interval

options.plotInterval : integer, interval between two calls of "plotnsga".

Description:

The overhead of plot in Matlab is very expensive. And it's not necessary to plot every generation for function optimization, a large interval value could speedup the optimization.

3.12. Parallel computation

options.useParallel : string, {'yes', 'no'}, specified if parallel computation is used.

options.poolsize : scalar, the number of worker processes. If you have a quat-core processor, *poolsize* could be set to 3, then you can do other things when the optimization is progressing.

Description:

The parallel computation is very useful when the evaluation of objective function is very time-expensive and you have a multicore/multiple processor(s) computer. If *options.useParallel* is specified as 'yes', the program would start multiple worker processes and use `parfor` to calculate each objective function (Parallel Computation Toolbox in Matlab is required). This procedure is showed in Fig. 3. Refer Matlab helps for details about parallel computation.

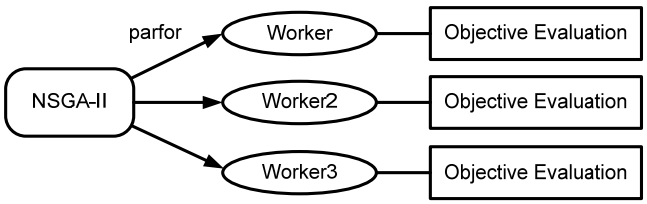


Fig. 3: Parallel computation in NGPM

Example:

```
options.useParallel = 'yes';
options.poolsize = 2;
```

4. R-NSGA-II: Reference-point-based NSGA-II

4.1. Introduction

The two objectives of multi-objective optimization are:

(1) Find the whole Pareto-optimal front.

(2) Get a well-distributed solution set in the front.

NSGA-II could do this well. But at last, only one or several solutions may be chose. Deb[2] proposed a modified procedure — R-NSGA-II — based on NSGA-II to get preference solutions by specified reference points. This procedure provides the decision-maker with a set of solutions near the preference solution(s), so that a better and a more reliable decision can be made.

4.2. Using the R-NSGA-II

The parameters below would be used in R-NSGA-II:

options.refPoints : matrix, Reference point(s) used to specify preference. Each row is a reference point in objective space.

options.refWeight : vector, weight factor used in the calculation of Euclidean distance. If no value is specified, all objectives have the same weight factor 1.0. It's the w_i in Eq. (4).

options.refEpsilon : scalar, a parameter used in epsilon-based selection strategy to control the spread of solution. All solutions having a weighted normalized Euclidean distance equal or less than ε would have a large preference distance in the next selection procedure. A large number (such as 0.01) would get a wide spread solution set near reference points, while a small value (such as 0.0001) would get a narrow spread solution set.

options. refUseNormDistance : string, {'front', 'ever', 'no'}, specify which approach

would be used to calculate the preference distance in R-NSGA-II.

"front" : (default) Use maximum and minimum objectives in the front as normalized factor. It means the $f_i^{\max}$ and $f_i^{\min}$ in Eq. (4) are the maximum and minimum objective values in the front.

"ever": Use maximum and minimum objectives ever found as normalized factor. It means the $f_i^{\max}$ and $f_i^{\min}$ in Eq. (4) are the maximum and minimum objective values ever found begin from the initialization population. In many test problems, it's similar to "front" parameter.

"no": Do not use normalized factor, only use Euclidean distance. It means $f_i^{\max} - f_i^{\min} = 1$.

$$d_{ij} = \sqrt{\sum_{i=1}^{M} w_i \left(\frac{f_i(\boldsymbol{x}) - \overline{z}_i}{f_i^{\max} - f_i^{\min}} \right)^2} \tag{4}$$

Example:

```
options.refPoints = [0.1 0.6; 0.3 0.6; 0.5 0.2; 0.7 0.2; 0.9 0;];
options.refWeight = [0.2 0.8];
options.refEpsilon = 0.001;
options.refUseNormDistance = 'no'
```

A test example ZDT1 can be find in "TP_R-NSGA2" folder.

5. Test Problems

5.1. TP1: KUR

File: TP_KUR.m, TP_KUR_fun.m

The KUR[1] problem has two objective function and no constraint except for bound constraints.

$$\begin{aligned} \min.\ & f_1(x_1,x_2,x_3) = \sum_{i=1}^{2}\left[-10\exp\left(-0.2\sqrt{x_i^2 + x_{i+1}^2}\right)\right] \\ & f_2(x_1,x_2,x_3) = \sum_{i=1}^{3}\left[|x_i|^{0.8} + 5\sin x_i^3\right] \\ \text{s.t.}\quad & -5 \le x_i \le 5, \quad i = 1,2,3 \end{aligned} \tag{5}$$

Some of the optimization parameters were showed in Table 2.

Table 2: Optimization parameters

Parameter	Value
Population size	50
Maximum generation	100
Crossover operator	Intermediate, ratio=1.2
Mutation operator	Gaussian, scale=0.1, shrink=0.5

Fig. 4 shows the whole population of generation 100.

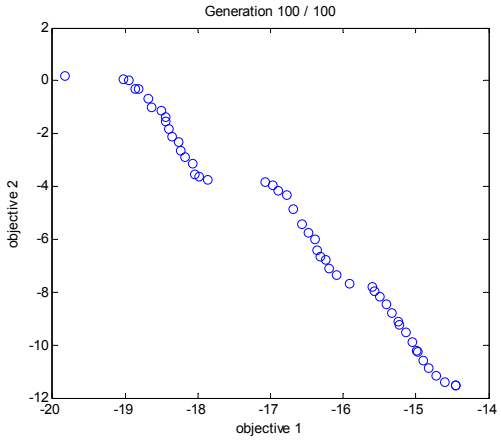


Fig. 4: The last population of problem KUR

5.2. TP2: TNK

File: TP_TNK.m, TP_TNK_objfun.m

The TNK[1] problem has two simple objective function and two complicated constraints except for bound constraints.

$$\begin{aligned} \min. \quad & f_1(x_1,x_2) = x_1 \\ & f_2(x_1,x_2) = x_2 \\ \text{s.t.} \quad & g_1(x) = -x_1^2 - x_2^2 + 1 + 0.1\cos(16\arctan(x_1 / x_2)) \le 0 \\ & g_2(x) = (x_1 - 0.5)^2 + (x_2 - 0.5)^2 \le 0.5 \\ & x_i \in [0,\pi], i = 1,2 \end{aligned} \tag{6}$$

The optimization parameters are showed in Table 2. Parallel computation is enabled, and *poolsize* is assigned 2. Actually, parallel computation is no essential here, and parallel computation cost more time then serial computation, since the overhead of interprocess communication exceeds the save time benefit from parallel computation. In my dual-core computer, the total time of parallel computation is 24.0s, while serial computation costs

12.5s.

Fig. 5 shows the whole population of generation 100.

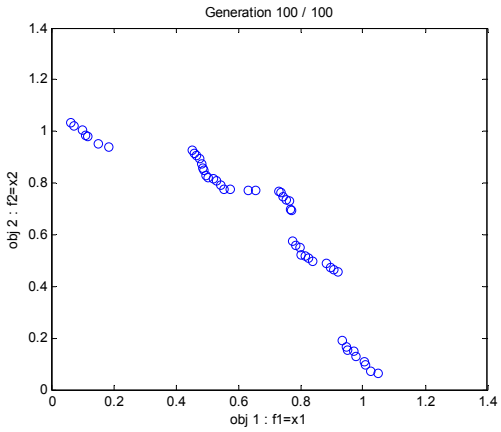


Fig. 5: The last population of problem TNK

5.3. TP3: Three-objective DTLZ2 — R-NSGA-II

File: TPR_DTLZ2_3obj.m, TPR_DTLZ2_objfun_3obj.m

The **DTLZ** [4] (Deb-Thiele-Laumanns-Zitzler) test problems are a set of MOPs for testing and comparing MOEAs. They are scalable to a user defined number of objectives.

$$
\begin{aligned}
\min. \quad & f_1(x) = (1+g(\boldsymbol{x}_M))\cos(x_1\pi/2)\cos(x_2\pi/2)\cdots\cos(x_{M-2}\pi/2)\cos(x_{M-1}\pi/2) \\
& f_2(x) = (1+g(\boldsymbol{x}_M))\cos(x_1\pi/2)\cos(x_2\pi/2)\cdots\cos(x_{M-2}\pi/2)\sin(x_{M-1}\pi/2) \\
& f_3(x) = (1+g(\boldsymbol{x}_M))\cos(x_1\pi/2)\cos(x_2\pi/2)\cdots\sin(x_{M-2}\pi/2) \\
& \vdots \\
& f_{M-1}(x) = (1+g(\boldsymbol{x}_M))\cos(x_1\pi/2)\sin(x_2\pi/2) \\
& f_M(x) = (1+g(\boldsymbol{x}_M))\sin(x_1\pi/2) \\
s.t. \quad & 0 \le x_i \le 1,\ i = 1,2,...,n \\
where \quad & g(\boldsymbol{x}_M) = \sum_{x_i \in \boldsymbol{x}_M} (x_i - 0.5)^2
\end{aligned}
\tag{7}
$$

where M is the number of objectives, n is the number of variables, $\boldsymbol{x}_M$ represents x_i for $i \in [M,n]$. It is recommended that $n = M + 9$. For DTLZ2 problem, Pareto-optimal solutions satisfy $\sum_{i=1}^{M} f_i^2 = 1$, and $x_i = 0.5$ for $i \in [M,n]$.

Here, optimization parameters below were used. There are two reference points: (0.2,0.2,0.6) and (0.8,0.6,1.0). The definition of ε is different from the Deb's definition, thus ε=0.002 was used instead of 0.01 in [2].

```
options.popsize = 200;          % populaion size
options.maxGen  = 200;          % max generation
```

```
options.refPoints = [0.2 0.2 0.6; 0.8 0.6 1.0];
options.refEpsilon = 0.002;
```

Fig. 6 shows the last population of three objective DTLZ2 problem and the true Pareto front.

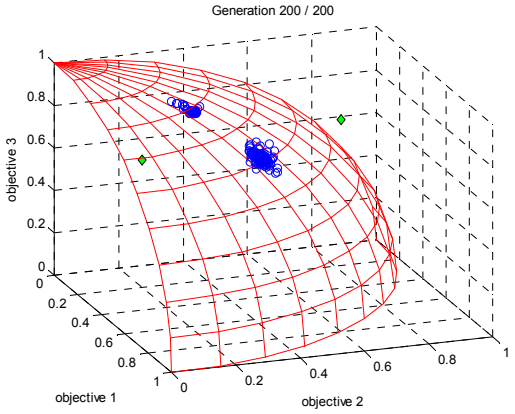


Fig. 6: The last population of three objective DTLZ2 problem

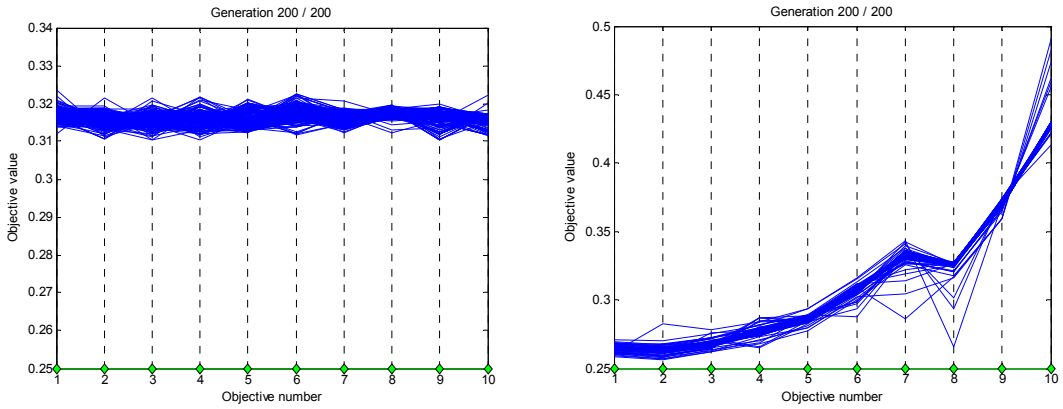
5.4. TP4: 10-objective DTLZ2 — R-NSGA-II

File: TPR_DTLZ2_10obj.m, TPR_DTLZ2_objfun_10obj.m

For 10-objective DTLZ2 problem, we used reference point: $f_i = 0.25$ for all $i = 1,2,...,10$. In Deb's paper[2] , it's said that the solution with f_i=0.316 is closest to the reference point. This is not true when normalized Euclidean distance is used: the points do not concentrates near f_i =0.316. If you want to get similar results as ref [2] , *options.refUseNormDistance* must be specified as 'no':

```
options.refUseNormDistance = 'no';
```

Then, similar result would be get as showed in Fig. 7(a). If *options.refUseNormDistance* was specified as default value 'front', you will get the result showed in Fig. 7(b).



(a) refUseNormDistance='no' (b) refUseNormDistance='front'

Fig. 7: 10-objective DTLZ2 problem with different refUseNormDistance parameter

6. Disclaimer

This code is distributed for academic purposes only. If you have any comments or find any bugs, please send an email to lsssswc@163.com.

7. Appendix A: Version history

v1.4 [2011-07-26]

1. Add: Support three or more objectives visualization display in "plotnsga".
2. Add: R-NSGA-II problem: DTLZ2.
3. Improve efficiency for large generation.

v1.3 [2011-07-15]

1. Add: Implement reference-point-based NSGA-II procedure -- R-NSGA-II.
2. Add: NSGA-II test problem: ZDT1, ZDT2, ZDT3 and ZDT6.
3. Improve: Improve the efficiency of "ndsort" function, get a 48% speedup for TP_CONSTR problem.
4. Improve: Save the output file ID to options structure for no explicit clear in optimization script file.
5. Modify: Modify the crossover and mutation strategy from individuals to variables.

v1.1 [2011-07-01]

1. Add: Load and plot population from previous optimization result file.
2. Add: Initialize population using exist optimization result or file.

v1.0 [2011-04-23]

The first version distributed.

Reference:

[1] Deb K, Pratap A, Agarwal S, et al. A fast and elitist multiobjective genetic algorithm NSGA-II[J]. Evolutionary Computation. 2002, 6(2): 182-197.

[2] Deb K, Sundar J, U B R N, et al. Reference point based multi-objective optimization using evolutionary algorithms[J]. International Journal of Computational Intelligence Research. 2006, 2(3): 273-286.

[3] Matlab Help, Global optimization toolbox.

[4] Deb K, Thiele L, Laumanns M, et al. Scalable Test Problems for Evolutionary Multi-Objective Optimization[C]. Piscataway, New Jersey: 2002.