Sections and Chapters

Gubert Farnsworth

ENME 625: Multi-Displinary Optimization
5/12/2017

Contents

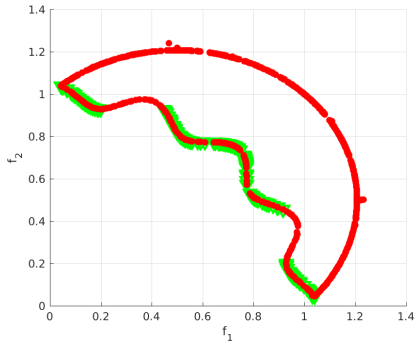

Figure 1: This is TNK for 20 chromosomes and 1000 runs

1 Introduction

This is the first section.

Unconstrained MOGA Problems

We used this textbook [1]

1.1 ZDT1

Talk about ZDT1 here.

Minimize $\mathrm{f}_1(\mathbf{x}) = X_1$

Maximize $f_2(\mathbf{x}) = g * h$

where $g = 1 + \frac{9}{(nvar-1)}. * sum_{i=2}^{n} var(X_i)$

$h = 1 - sqrt\frac{f_1}{g}$

$nvar = 30$

subject to$\mathbf{X} = \mathbf{X} \leq 1$

$\mathbf{X} = \mathbf{X} \geq 0$

2 Constrained MOGA Problems

This is a reference to Azarms constraint paper [2].

2.1 TNK

Here is figure 1.

References

[1] K. Deb, "Multi-objective optimization using evolutionary algorithms, 2001," *Chicheter, John-Wiley.*, 2001.

[2] A. Kurpati, S. Azarm, and J. Wu, "Constraint handling improvements for multiobjective genetic algorithms," *Structural and Multidisciplinary Optimization*, vol. 23, no. 3, pp. 204–213, 2002.

3 Appendix

Matlab Code

```
function [optF]=MasterCode(prob,nChrome,nRun,save_figure,
    use_matlabs_moga)
% load .mat file
results_and_params = load('results_and_params.mat');

%clear all;
% close all;
%clc;
warning off

if(nargin < 1)
    prompt = 'Which Test Problem Do You Want To Run? \n 1
        - ZDT1\n 2 - ZDT2 \n 3 - ZDT3 \n 4 - OSY \n 5 -
       TNK \n 6 - CTP \n';
    prob = input(prompt);
end
if nargin <2
    prompt2 = 'How Many Chromosomes? Suggest 20-30 ';
    nChrome = input(prompt2);
end
if nargin <3
    prompt3 = 'How Many Runs? Suggest >40: ';
    nRun = input(prompt3);
end
if nargin <4
    prompt4 = 'Autosave figures [ 1 or 0 ]?';
    save_figure=input(prompt4);
end
if nargin <5
    prompt5 = 'Use Matlabs MOGA [ 1 or 0 ]?';
    use_matlabs_moga=input(prompt5);
end
```

```
30
31 %this_problem = results_and_params(prob,1);
32 %%
33 %prob=1; nChrome = 1; nRun = 40;
34
35
36
37 % ZD-func is our problem function
38 switch prob
39     case 1
40         problem_function = @(X) ZDT1(X);
41         nvar = 30; LB = zeros(1,nvar); UB = ones(1,nvar);
42         problem_contraints = [];
43     case 2
44         problem_function = @(X) ZDT2(X);
45         nvar = 30; LB = zeros(1,nvar); UB = ones(1,nvar);
46         problem_contraints = [];
47     case 3
48         problem_function = @(X) ZDT3(X);
49         nvar = 30; LB = zeros(1,nvar); UB = ones(1,nvar);
50         problem_contraints = [];
51     case 4
52         problem_function = @(X) OSY(X);
53         nvar = 6; LB = [0,0,1,0,1,0]; UB =
               [10,10,5,6,5,10];
54         problem_contraints  =@(X) OSY_constraints(X);
55     case 5
56         problem_function = @(X) TNK(X);
57         nvar = 2; LB = [0,0]; UB=[pi,pi];
58         problem_contraints  =@(X) TNK_constraints(X);
59     case 6
60         problem_function = @(X) CTP(X);
61         nvar = 10; LB = -5*ones(1,10); UB = 5*ones(1,10);
               LB(1,1) = 0; UB(1,1) = 1;
62         problem_contraints  =@(X) CTP_constraints(X);
63     otherwise
64         problem_function = @(X) 0;
65 end
66
67 A = []; b = []; Aeq = []; beq = [];
68
69 if use_matlabs_moga ==1
70     % Initilize Population
71     %Initialize the population based on the given lower
           and upper bounds. Use
72     %MATLABs random number generator.
```

```
73      % Start with the default options
74      options = optimoptions('gamultiobj');
75      % Modify options setting
76      options = optimoptions(options,'PopulationSize', nRun
            );
77      options = optimoptions(options,'CrossoverFcn',
            @crossoverscattered);
78      options = optimoptions(options,'Display', 'final');
79      options = optimoptions(options,'PlotFcn', {
            @gaplotpareto });
80      options = optimoptions(options,'ParetoFraction', 0.9)
            ;
81      [~,optF] = gamultiobj(problem_function,nvar
            ,[],[],[],[],LB,UB,problem_contraints,options);
82      return
83  end
84
85  Pareto = [];
86  options = optimoptions(@ga,'PopulationSize',nChrome,'
        UseVectorized',true);
87  %options.FunctionTolerance = 0.001*options.
        FunctionTolerance
88
89  optF =[];
90  for gen = 1:nRun
91      %Obj_fcn = @(X) fitFCN8(X,problem_function,optF);
92      Obj_fcn = @(X) fitFCN5(X,problem_function);
93      [X,fval,exitflag,output] = ga(Obj_fcn,nvar,A,b,Aeq,
            beq,LB,UB,[],options);
94      [optF(gen,:)] = problem_function(X);
95      optX(gen,:) = X;
96  end
97  nfunc = optF(1,end-2);
98
99   P = paretoset(optF(:,1:nfunc));
100  m = 1;
101     for k = 1:length(P)
102         if P(k) == 1
103             Pareto(m,:) = optF(k,1:2); m = m+1;
104         end
105     end
106
107 figure
108 hold on;
109 plot(Pareto(:,1),Pareto(:,2),'gv','LineWidth',2,'
        MarkerSize',10);
```

```
110 plot(optF(:,1),optF(:,2),'r*','LineWidth',2)
111 hold on; grid on;
112 xlabel('f_1'); ylabel('f_2')
113 handle = gcf;
114 if save_figure == 1
115     %Save the figures
116     dir_val = pwd;
117     saveFigure(handle,[dir_val,dir_val(1),num2str(prob),'
        _',num2str(nChrome,'%03.0f'),'_',num2str(nRun,'
        %04.0f')]);
118     print([dir_val,dir_val(1),num2str(prob),'_',num2str(
        nChrome,'%03.0f'),'_',num2str(nRun,'%04.0f'),'.png
        '],'-dpng');
119
120     %Save the .mat file
121     %this_problem = struct('prob',prob,'nChrome',nChrome
        ,'nRun',nRun,'optF',optF,'Matlabs')
122     this_problem.prob = prob;
123     this_problem.nChrome
124     results_and_params{prob,1} = this_problem;
125     save('results_and_params.mat',results_and_params)
```

```
1  function [ fit ] = fitFCN5(X, ZD_func)
2  %NSGA algorithm. Use Approach 1 for sorting
3
4  func = ZD_func(X);
5
6  %
7  % if isempty(existing_points)
8  %     fit = sum(ZD_func(X));
9  %     return
10 % end
11
12 %% Find Dominate Points
13
14 % Modify existing code to find Pareto points to find
       dominant layers
15
16 % func = [existing_points;ZD_func(X)];
17 nfunc = func(1,end-2);
18 nconstr = func(1,end-1);
19 g = func(:,nfunc+1:nfunc+nconstr);
20 nconstr_lin = func(1,end);
21 h = func(:,nfunc+nconstr+1:nfunc+nconstr+nconstr_lin);
22 func = func(:,1:nfunc);
23
```

```
24 [M,~] = size(X);
25
26
27
28
29 if nconstr == 0 && nconstr_lin ==0
30     nc_col = nfunc + 3;
31     init_fit_col = nfunc + 4;
32     sim_col = nfunc+5;
33     indecies = [1:M]';
34     func = [func,indecies]; %We need to know indecies
           later so this should save time
35     P_temp = func;
36     level = 0;
37     level_col = nfunc +2;
38     func(:,level_col) = 0;
39     while ~isempty(P_temp)
40         level = level+1;% increment the level value
41         if length(P_temp(:,1))== 1
42             %If there is only one value left at the end,
                   assign this to a level
43             func(P_temp(:,nfunc+1),level_col) = level;
44             break
45         end
46         place = paretoset(P_temp(:,1:nfunc)); % get all
               the indecies in the lowest layer
47         for k = 1:length(place)
48             if place(k) == 1
49                 current_level_indecies = P_temp(k,nfunc
                       +1); %map them from
50                 func(current_level_indecies, level_col) =
                       level; % assiged from prtp
51             end
52         end
53
54         P_temp(place, :) = [];
55     end
56
57     numLayer = level;
58
59     %Make sure all individuals have a layer number
60     flag = 0;
61     for k = 1:M
62         if func(level_col)==0
63             func(k,level_col) = numLayer+1;
64             flag = 1;
```

```
65          end
66      end
67      if flag == 1, numLayer = numLayer+1; end
68
69      %% Similarity
70      %Assess similarity layer-by-layer, assess in
            objective space.
71
72      %sigma = 0.75;
73      sigma = 0.158;
74      %epsilon = 0.25;
75      epsilon = 0.22;
76      alpha = 1;
77      var_rem = 0;
78      F_min = M+epsilon;
79      for k = 1:numLayer
80          Fitness = []; incl = [];
81          incl = find(func(:,level_col)==k); % incl =
                include
82          Fitness = func(incl,1:nfunc);
83          var_rem = var_rem+length(Fitness(:,1));
84          if(isempty(Fitness) == 0)
85              if length(incl) == 1
86
87                  F_int = F_min-epsilon;
88                  Fit_share = F_int;
89                  func(incl,sim_col+1) = F_int;
90                  func(incl,sim_col) = F_int;
91                  func(incl,nc_col) = 1;
92              else
93                  for m = 1:nfunc
94                      maxF(m) = max(Fitness(:,m));
95                      minF(m) = min(Fitness(:,m));
96                      func(m,init_fit_col) = minF(m);
97                  end
98                  d = []; similar = []; sh = [];
99                  for i = 1:length(incl)
100                     F_int = F_min-epsilon;
101                     for j = 1:length(incl)
102                         for p = 1:nfunc
103                             similar(p) = ((Fitness(i,p)-
                                    Fitness(j,p))/(maxF(p)-
                                    minF(p)))^2;
104                         end
105                         d(i,j) = sqrt(sum(similar));
106                         if d(i,j)<=sigma
```

```
107                                 sh(i,j) = 1-(d(i,j)/sigma)^
                                        alpha;
108                             else sh(i,j) = 0;
109                             end
110                         end
111                         nc(i) = sum(sh(i,:));
112                         Fit_share(i) = F_int/nc(i);
113                         func(incl(i),nc_col) = nc(i);
114                         func(incl(i),sim_col) = Fit_share(i);
115                         func(incl(i),sim_col+1) = F_int;
116
117                     end
118                 end
119                 F_min = min(Fit_share);
120             end
121
122         end
123
124         %Since a greater fitness value is a larger number, we
              use the inverse
125         fit = -func(:,sim_col);
126
127     %% Constraint Handling
128     else
129         Cmax = 1.2; Cmin = 0.8; r = 0.8*M;
130         %CF1 = 0.0005+(0.015-0.0005)*rand;
131         %CF2 = 0.0005+(0.015-0.0005)*rand;
132         CF1 = 0.01;
133         CF2 = 0.01;
134         rank = zeros(1,M);
135
136         % Assign moderate rank to all feasible solutions
137          for k = 1:M
138             flag = 0; flag_lin = 0;
139             for p = 1:nconstr
140                 if g(k,p)>0, flag = 1;
141                 end
142                 if nconstr_lin ~= 0
143                     if h(k,p)~=0, flag_lin = 1;
144                     end
145                 end
146             end
147             if flag == 0 && flag_lin == 0
148                 rank(k) = 0.5*M;
149             end
150          end
```

```
151
152         % Collect together feasible population
153         feas_pop = []; infeas_pop = []; m = 1;
154         for k = 1:M
155             if rank(k) ~= 0
156                 if isempty(h)
157                     feas_pop = [feas_pop;func(k,:),g(k,:)];
158                 else
159                     feas_pop = [feas_pop;func(k,:),g(k,:),h(
                           k,:)];
160                 end
161             else
162                  if isempty(h)
163                      infeas_pop = [infeas_pop;func(k,:),g(k
                           ,:)];
164                  else
165                      infeas_pop = [infeas_pop;func(k,:),g(k
                           ,:),h(k,:)];
166                  end
167                  loc(m) = k; m = m+1; %keep track of which
                       solutions were infeasible
168             end
169         end
170
171         % Identify noninferior points
172         if ~isempty(feas_pop)
173              place = paretoset(feas_pop(:,1:nfunc));
174              m = 1;
175              for k = 1:length(place)
176                  if place(k) == 1
177                      Pareto(m,:) = feas_pop(k,:); %Assign
                           noninferior points along with
                           constraint values
178                      rank(k) = 1; m = m+1;
179                  end
180              end
181         end
182        % Evaluate rank for infeasible individuals
183        g = infeas_pop(:,nfunc+1:nconstr+nfunc);
184        h = infeas_pop(:,nconstr+nfunc+1:end);
185
186        for k = 1:length(g(:,1))
187            for p = 1:nconstr
188                if g(k,p)<=0
189                    feas_g(k,p) = 0; delta_g(k,p) = 0;
190                else
```

```
                feas_g(k,p) = g(k,p); delta_g(k,p) = 1;
            end
        end
        if nconstr_lin == 0
            feas_h = zeros(length(g(:,1)),1);
            delta_h = zeros(length(g(:,1)),1);
        else
            for n = 1:nconstr_lin
                feas_h(k,n) = abs(h(k,n));
            end
            if h(k,p)==0, delta_h(k,p) = 0;
            else delta_h(k,p) = 1;
            end
        end
    end
    num1 = sum(feas_g,2)+sum(feas_h,2);
    denom1 = (sum(sum(feas_g))+sum(sum(feas_h)))/M;
    J = nconstr; K = nconstr_lin;
    num2 = (sum(delta_g,2)+sum(delta_h,2));
    denom2 = (J+K);

    factor1 = CF1.*(num1./denom1);
    factor2 = CF2.*(num2./denom2);



    for k = 1:length(g(:,1))
        if (factor1(k)> mean(factor1)) && (factor2(k) <
            mean(factor2))
            w1 = 0.75; w2 = 0.25;
        elseif (factor1(k) < mean(factor1)) && (factor2(k
            ) > mean(factor2))
            w1 = 0.25; w2 = 0.75;
        else
            w1 = 0.5; w2 = 0.5;
        end
        fit_constr(k) = -((Cmax-(Cmax-Cmin)*(r-1)/(M-1))
            -(w1.*factor1(k)+w2.*factor2(k)));
    end

    for k = 1:length(loc)
        rank(loc(k)) = fit_constr(k);
    end
    fit = rank;
```

```
234 end
235 end
```