# Παράρτημα: Πηγαίος κώδικας αλγορίθμων

```
{*****************************************************************************
 *****************************************************************************
          ========================================
          THE EVOLUTIONARY ANNEALING-SIMPLEX METHOD
          ========================================
*****************************************************************************
*****************************************************************************}
procedure AnnealSimplex(n, m: Integer; var xl, xu, xmin, xmax, xopt:
TArrayOfDouble; var_rigid: TBoolArray; F: TMathFuncMult; var fopt: Double; var
eval: Integer; ftol: Double; maxeval: Integer; ratio, pmut, beta: Double;
maxclimbs: Integer; iniflg, restart: Boolean; var StopOptim: Boolean);

{*****************************************************************************
Multidimensional minimization of the function F(x), where x[1..n] is a vector in
n dimensions, by simulated annealing combined with the downhill simplex method
of Nelder and Mead. Input arguments are:

n      = problem dimension
m      = population count (m > n + 1)
xmin[n]   = lower parameter bounds
xmax[n]   = upper parameter bounds
xopt[n]   = vector containing the optimal values of control variables
F      = objective function
fopt    = the minimal value of the objective function
eval    = number of function evaluations
ftol    = the fractional convergence tolerance to be achieved in the
       function value for an early return
maxeval   = maximum number of function evaluations
ratio    = fraction of temperature reduction, when a local minimum is found
beta    = annealing schedule parameter
pmut    = probability of accepting an offspring generated via mutation
maxclimbs  = maximum number of uphill steps
iniflg   = if true then xopt is one of the starting points
restart   = if true then the search procedure is repeated
StopOptim  = flag for terminating optimization
*****************************************************************************}

var
 pop: T2DArrayOfDouble; {population of m points}
 fpop: TArrayOfDouble; {population fitness}
 simplex: T2DArrayOfDouble; {sub-population of n+1 points - simplex vertices}
 fsimplex: TArrayOfDouble; {simplex fitness}
 pstart, pref, ptry, pcent: TArrayOfDouble; {temp arrays}
 xmean, xstdev: TArrayOfDouble; {arrays of population statistics}
 ipos: TIntArray; {index array}
 fmin, fmax: Real; {minimum and maximum criteria values}
 fmean, fstdev: Real; {statistics}
 temperature: Real; {control parameter}
 imin, imax, lpos: Integer;
 i, ihi, ilo, istart, j, step: Integer;
```

```
 rtol, sum, yhi, ylo, ystart, ytry, yref, ylast, fac, maxdist: Real;
 BoundReached: Boolean;
 BestStep: Integer;
 RejectPoint: Boolean;
 label 100, 200, 999;

begin
 {Check for consistency of input arguments}
 If (m<n+1) then m:=n+1;

 {Create arrays}
 SetLength(pop, m, n);
 SetLength(fpop, m);
 SetLength(simplex, n+1, n);
 SetLength(fsimplex, n+1);
 SetLength(xmean, n);
 SetLength(xstdev, n);
 SetLength(ptry, n);
 SetLength(pcent, n);
 SetLength(pstart, n);
 SetLength(pref, n);
 SetLength(ipos, n+1);

 eval:=0;
 fopt:=1e100;

100:
 fmin:=1e100;
 fmax:=-fmin;

 {Initialization}
 If iniflg then
 begin
  For j:=0 to n-1 do pop[0,j]:=xopt[j];
  fpop[0]:=F(xopt);
  fmin:=fpop[0]; fmax:=fpop[0];
  i:=1;
 end
 else i:=0;

 Repeat
  For j:=0 to n-1 do ptry[j]:=xmin[j]+random*(xmax[j]-xmin[j]);
  ytry:=F(ptry);
  For j:=0 to n-1 do pop[i,j]:=ptry[j];
  fpop[i]:=ytry;
  If ytry<fmin then fmin:=ytry;
  If ytry>fmax then fmax:=ytry;
  i:=i+1;
 until (i=m);

 temperature:=fmax-fmin;
 eval:=eval+m;

 {*****************}
 {*** Main loop ***}
 {*****************}

 Repeat
```

```
If StopOptim then goto 999;

{Compute the statistics of the population}
For j:=0 to n-1 do
begin
 sum:=0;
 For i:=0 to m-1 do sum:=sum+pop[i,j];
 xmean[j]:=sum/m;
 sum:=0;
 For i:=0 to m-1 do sum:=sum+(pop[i,j]-xmean[j])*(pop[i,j]-xmean[j]);
 xstdev[j]:=sqrt(sum/m);
end;
sum:=0;
For i:=0 to m-1 do sum:=sum+fpop[i];
fmean:=sum/m;
sum:=0;
For i:=0 to m-1 do sum:=sum+(fpop[i]-fmean)*(fpop[i]-fmean);
fstdev:=sqrt(sum/m);

{Generate a simplex, selecting its vertices randomly from the actual
 population}
If (m=n+1) then for i:=0 to m-1 do ipos[i]:=i
else
begin
 ipos[0]:=Round(random*(m-1));
 For i:=1 to n do
 begin
  200: lpos:=Round(random*(m-1));
  For j:=0 to i do if lpos=ipos[j] then goto 200;
  ipos[i]:=lpos;
 end;
end;

{Assign the coordinates and the function value to each vertex}
For i:=0 to n do
begin
 lpos:=ipos[i];
 For j:=0 to n-1 do simplex[i,j]:=pop[lpos,j];
 fsimplex[i]:=fpop[lpos];
end;

{Determine the highest (worst) and the lowest (best) point, as well as the
 randomized worst point, according to the criterio xw = max f(x) + rnd*T}
ilo:=0; ihi:=0; istart:=0;
For i:=1 to n do if fsimplex[i]<fsimplex[ilo] then ilo:=i else
         if fsimplex[i]>fsimplex[ihi] then ihi:=i;
yhi:=fsimplex[ihi]; ylo:=fsimplex[ilo];

If temperature>(beta*(yhi-ylo)) then temperature:=beta*(yhi-ylo);

ystart:=ylo;
For i:=0 to n do
begin
 ytry:=fsimplex[i]+random*temperature;
 If (ytry>ystart) and (i<>ilo) then
 begin
  istart:=i;
```

```
    ystart:=ytry;
   end;
  end;
  ystart:=fsimplex[istart];

  {Store the randomized worst vertex}
  For i:=0 to n-1 do pstart[i]:=simplex[istart,i];

  {Compute the centroid of the simplex}
  For i:=0 to n-1 do
  begin
   sum:=0;
   For j:=0 to n do sum:=sum+simplex[j,i];
   pcent[i]:=(sum-pstart[i])/n;
  end;

  {Make a reflection step}
  fac:=0.5+random;
  NewPoint(n, eval, 1+fac, -fac, F, pref, pcent, pstart, xl, xu, xmin, xmax,
var_rigid, yref, BoundReached);
  If StopOptim then goto 999;

  {If the reflection point is either not accepted (no move) or fr<fw (downhill
   move) the method follows the modified (quasi-stochastic) Nelder-Mead
   procedure, making contraction and expansion moves respectively}

  If yref<ystart then
  begin

   {Accept the reflection point}
   For i:=0 to n-1 do simplex[istart,i]:=pref[i];
   fsimplex[istart]:=yref;
   lpos:=ipos[istart];
   For i:=0 to n-1 do pop[lpos,i]:=pref[i];
   fpop[lpos]:=yref;

   {If the reflected point is better than the lowest, try a line minimization
    employing subsequent random expansion steps, else try an outside contraction
    step between xc and xr}
   If (yref<ylo) and (not BoundReached) then
   {Multiple expansion}
   Repeat
    fac:=fac+0.5+random;
    NewPoint(n, eval, 1+fac, -fac, F, ptry, pcent, pstart, xl, xu, xmin, xmax,
var_rigid, ytry, BoundReached);
    If StopOptim then goto 999;
    If ytry<fsimplex[istart] then
    begin
     fsimplex[istart]:=ytry;
     For j:=0 to n-1 do simplex[istart,j]:=ptry[j];
     lpos:=ipos[istart];
     For j:=0 to n-1 do pop[lpos,j]:=ptry[j];
     fpop[lpos]:=ytry;
    end;
   until (ytry>yref) or BoundReached
   else
   {Outside contraction}
   begin
```

```
    fac:=random*fac;
    NewPoint(n, eval, 1+fac, -fac, F, ptry, pcent, pstart, xl, xu, xmin, xmax,
var_rigid, ytry, BoundReached);
    If StopOptim then goto 999;
    If ytry<fsimplex[istart] then
    begin
     fsimplex[istart]:=ytry;
     For j:=0 to n-1 do simplex[istart,j]:=ptry[j];
     lpos:=ipos[istart];
     For j:=0 to n-1 do pop[lpos,j]:=ptry[j];
     fpop[lpos]:=ytry;
    end;
   end; {outside contraction step}

  end
  else if (yref-random*temperature)>(ystart+random*temperature)
  then
  begin

   {Don't accept the reflection step and try an inside contraction step}
   fac:=random;
   NewPoint(n, eval, fac, 1-fac, F, ptry, pcent, pstart, xl, xu, xmin, xmax,
var_rigid, ytry, BoundReached);
   If StopOptim then goto 999;
   If ytry<fsimplex[istart] then
   begin
    fsimplex[istart]:=ytry;
    For j:=0 to n-1 do simplex[istart,j]:=ptry[j];
    lpos:=ipos[istart];
    For j:=0 to n-1 do pop[lpos,j]:=ptry[j];
    fpop[lpos]:=ytry;
   end;

   {Reduce the temperature}
   temperature:=ratio*temperature;

   {Multiple contraction step}
   If ytry>ystart then
   begin
    For i:=0 to n-1 do pstart[i]:=simplex[ilo,i]; {store the coordinates of the
best vertex}
    For i:=0 to n do if i<>ilo then
    begin
     For j:=0 to n-1 do ptry[j]:=simplex[i,j]; {store the coordinates of the ith
vertex}
     fac:=0.5;
     NewPoint(n, eval, fac, fac, F, ptry, ptry, pstart, xl, xu, xmin, xmax,
var_rigid, ytry, BoundReached);
     If StopOptim then goto 999;
     lpos:=ipos[i];
     For j:=0 to n-1 do pop[lpos,j]:=ptry[j];
     fpop[lpos]:=ytry;
    end;
   end;
  end
  else
  begin
```

```
   {Accept the reflection point}
   For j:=0 to n-1 do simplex[istart,j]:=pref[j];
   fsimplex[istart]:=yref;
   lpos:=ipos[istart];
   For j:=0 to n-1 do pop[lpos,j]:=pref[j];
   fpop[lpos]:=yref;

   {Store the reflection point to the temporary vector pstart to use it for the
computation of the direction xr-xc}
   For i:=0 to n-1 do pstart[i]:=pref[i];

   {Try some random uphill steps along the reflection direction and store the best
of them; if you observe a hill climbing, replace the reflection point.
    Whenever you reach the bounds, exit from the search procedure}
   yref:=1e20;
   BoundReached:=False;
   step:=0; fac:=1;
   ylast:=yref;
   Repeat
    step:=step+1;
    fac:=fac+0.5+random;
    NewPoint(n, eval, 1+fac, -fac, F, ptry, pcent, pstart, xl, xu, xmin, xmax,
var_rigid, ytry, BoundReached);
    If StopOptim then goto 999;
    If ytry<ylast then
    begin
     BestStep:=step;
     yref:=ytry;
     For j:=0 to n-1 do pref[j]:=ptry[j];
    end
    else ylast:=ytry;
   until (step=maxclimbs) or BoundReached;

   {If any hill climbing occurs, try a mutation step by generating a random
    point out of the range (xmean-xstdev, xmean+xstdev)}
   If (BestStep>1) or (yref<fsimplex[istart]) then
   begin
    For j:=0 to n-1 do simplex[istart,j]:=pref[j];
    fsimplex[istart]:=yref;
    For j:=0 to n-1 do pop[lpos,j]:=pref[j];
    fpop[lpos]:=yref;
   end
   else
   begin
    {Mutation}
    For i:=0 to n-1 do
    begin
     If random>0.5 then
     begin
      maxdist:=xmax[i]-xmean[i]-xstdev[i];
      If maxdist<0 then maxdist:=0;
      ptry[i]:=xmean[i]+xstdev[i]+random*maxdist;
     end
     else
     begin
      maxdist:=xmean[i]-xstdev[i]-xmin[i];
      If maxdist<0 then maxdist:=0;
      ptry[i]:=xmean[i]-xstdev[i]-random*maxdist;
```

```
     end;
     if ptry[i]>xmax[i] then ptry[i] := xmax[i];
     if ptry[i]<xmin[i] then ptry[i] := xmin[i];
    end;
    ytry:=F(ptry);
    eval:=eval+1;
    If (ytry<fsimplex[istart]) or (random<pmut) then
    begin
     fsimplex[istart]:=ytry;
     For j:=0 to n-1 do simplex[istart,j]:=ptry[j];
     For j:=0 to n-1 do pop[lpos,j]:=ptry[j];
     fpop[lpos]:=ytry;
    end;
   end;
  end;

  {Determine the best and worst point into the population}
  imin:=0; imax:=0;
  For i:=1 to m-1 do
  begin
   If fpop[i]<fpop[imin] then imin:=i else
   If fpop[i]>fpop[imax] then imax:=i;
  end;

  fmin:=fpop[imin];
  fmax:=fpop[imax];

  {Check convergence criteria}
  If restart then
  begin
   try rtol:=2*ABS(fmax-fmin)/(ABS(fmax)+ABS(fmin))
   except rtol:=2*ABS(fmax-fmin)
   end;
   If rtol<ftol then Break;
  end;

 until (eval>=maxeval);

999:
 {Save the optimal solution}
 For i:=0 to n-1 do xopt[i]:=pop[imin,i];
 fopt:=F(xopt);

 {Destoy all arrays}
 pop:=nil; fpop:=nil; simplex:=nil; fsimplex:=nil; pstart:=nil; pref:=nil;
 ptry:=nil; pcent:=nil; xmean:=nil; xstdev:=nil; ipos:=nil;

end; {procedure AnnealSimplex}


procedure NewPoint(num_var: Integer; var eval: Integer; a1, a2: Real; F:
TMathFuncMult; var x, x1, x2, xl, xu, xmin, xmax: TArrayOfDouble; var_rigid:
TBoolArray; var fx: Real; var boundary: Boolean);
{Generates a feasible point according to the formula x[i] = a1*x1[i] + a2*x2[i]}

var
 i: Integer;
```

```
begin
 boundary:=False;
 For i:=0 to num_var-1 do
 begin
  x[i]:=a1*x1[i]+a2*x2[i];
  If var_rigid[i] then
  begin
   If x[i]<xmin[i] then
   begin
    x[i]:=xmin[i]; boundary:=True;
   end
   else if x[i]>xmax[i] then
   begin
    x[i]:=xmax[i]; boundary:=True;
   end;
  end
  else
  begin
   If x[i]<xl[i] then
   begin
    x[i]:=xl[i]; boundary:=True;
   end
   else if x[i]>xu[i] then
   begin
    x[i]:=xu[i]; boundary:=True;
   end;
  end;
 end;
 fx:=F(x);
 eval:=eval+1;
end; {procedure NewPoint}



{*****************************************************************************
 *****************************************************************************
       ========================================================
       THE MULTIOBJECTIVE EVOLUTIONARY ANNEALING-SIMPLEX METHOD
       ========================================================
*****************************************************************************
*****************************************************************************}
procedure MEAS(num_var, num_obj: Integer; var x_lower, x_upper, low_bound,
up_bound, weights, f_limit: TArrayOfDouble;
        F: TVectorMathFuncMult; var xopt, fopt: T2DArrayOfDouble; var minmax,
var_rigid: TBoolArray;
        pop_size: Integer; var eval, domin_count: Integer; ftol: Double; maxeval:
Integer; ratio, p_mutation, beta: Double);

{*****************************************************************************
Input arguments:
 num_var         = number of control variables
 num_obj         = number of objectives
 pop_size        = population size
 x_lower[num_var]    = lower parameter bounds
 x_upper[num_var]    = upper parameter bounds
 weights[num_obj]    = weights assigned to fitness
 f_limit[num_obj]    = boundary of feasible objective space
 var_rigid[num_var]   = true, if upper limits refer to physical constraints
```

```
 F[num_obj]       = vector objective function
 minmax[num_obj]     = min or max flag
 xopt[pop_size, num_var] = array of Pareto set
 fopt[pop_size, num_obj] = array of Pareto front
 eval          = number of function evaluations
 domin_count       = number of non-dominated points
 ftol          = the fractional convergence tolerance to be achieved in the
function value for an early return
 maxeval         = maximum number of function evaluations
 ratio          = fraction of temperature reduction, when a local minimum is found
 beta          = annealing schedule parameter
 p_mutation       = probability of accepting an offspring generated via mutation
 maxclimbs        = maximum number of allowed uphill steps
*************************************************************************}

var
 pop: my_population; {population of pop_size points}
 dominated_id: TIntArray; {array containing the id's of dominated individuals}
 feasible_count: Integer;
 callF: Boolean;
 simplex : my_population; {sub-population of num_var+1 points}
 xw   : TArrayOfDouble; {"worst" vertex, according to the probabilistic criterio}
 xcent  : TArrayOfDouble; {simplex centroid}
 smin, smax, sw: Integer;
 fw   : Double;     {dum_fitness of the worst vertex}
 pref, pexp, pcontr, pmut: Double;
 fref, fexp, fcontr, fmut: Double;
 pmin, pmax, pw: Double;
 tempind: my_individual;
 pstart, ptry: TArrayOfDouble; {temp arrays}
 xmean, xstdev: TArrayOfDouble; {arrays of population statistics}
 ipos: TIntArray; {index array}
 min_fitness, max_fitness: Double;
 fmin, fmax: TArrayOfDouble; {minimum and maximum dummy function values}
 temperature: Double; {control parameter}
 i, j, k: Integer;
 imin, imax, lpos: Integer;
 step: Integer;
 rtol, sum, ystart, ytry, yref, ylast, fac: Double;
 BoundReached: Boolean;
 ZeroReplaced: Boolean;
 division : Integer;
 label 100, 200;

begin

 division:=10;

 {Create arrays}
 SetLength(pop, pop_size);
 SetLength(dominated_id, pop_size);

 SetLength(ipos, num_var+1);
 SetLength(simplex, num_var+1);
 SetLength(xcent, num_var);
 SetLength(xw, num_var);

 SetLength(ptry, num_var);
```

```
 SetLength(pstart, num_var);

 SetLength(fmin, num_obj);
 SetLength(fmax, num_obj);
 SetLength(xmean, num_var);
 SetLength(xstdev, num_var);

 SetLength(tempind.x, num_var);
 SetLength(tempind.obj, num_obj);

 {Generate initial population}
 For i:=0 to pop_size-1 do with pop[i] do
 begin
  SetLength(x, num_var);
  SetLength(obj, num_obj);
  If i=0 then
  begin
   For j:=0 to num_var-1 do x[j]:=x_lower[j]+random*(x_upper[j]-x_lower[j]);
   callF:=F(x, obj);
  end
  else
  begin
   For j:=0 to num_var-1 do x[j]:=x_lower[j]+random*(x_upper[j]-x_lower[j]);
   callF:=F(x, obj);
  end;
  rank:=0; density:=0; fitness:=0; inferior_count:=0; superior_count:=0;
min_xdist:=0; min_fdist:=0;
 end;
 eval:=pop_size; ZeroReplaced:=False;

 CalcStatistics(pop_size, num_obj, num_var, fmin, fmax, imin, imax, min_fitness,
max_fitness, xmean, xstdev, pop);
 UpdateGrid(pop_size, num_obj, division, pop, fmin, fmax);
 CalcFitness(pop_size, num_obj, minmax, f_limit, dominated_id, domin_count,
feasible_count, pop);
 CalcStatistics(pop_size, num_obj, num_var, fmin, fmax, imin, imax, min_fitness,
max_fitness, xmean, xstdev, pop);
 temperature:=max_fitness-min_fitness;

 Repeat

  {Generate a simplex, selecting its vertices randomly from the actual population}
  If (pop_size=num_var+1) then for i:=0 to pop_size-1 do ipos[i]:=i
  else
  begin
   {Pick up at least one vertex from the dominated set}
   If domin_count>0 then
   begin
    j:=RandomRange(0, domin_count-1);
    ipos[0]:=dominated_id[j];
   end else ipos[0]:=Round(random*(pop_size-1));

   For i:=1 to num_var do
   begin
    200: lpos:=RandomRange(0, pop_size-1);
    For j:=0 to i do if lpos=ipos[j] then goto 200;
    ipos[i]:=lpos;
   end;
```

```
  end;

  {Copy the parents from the population to the simplex-based mating pool
   and compute dum_fitness = f(s) = p(s) + u*T}
  For i:=0 to num_var do
  begin
   lpos:=ipos[i];
   simplex[i]:=pop[lpos];
   simplex[i].dum_fitness:=simplex[i].fitness+random*temperature;
  end;

  {Determine the real highest (worst) and the lowest (best) parent within simplex}
  smin:=0; smax:=0;
  For i:=1 to num_var do if simplex[i].fitness<simplex[smin].fitness then smin:=i
else if simplex[i].fitness>simplex[smax].fitness then smax:=i;
  pmin:=simplex[smin].fitness; pmax:=simplex[smax].fitness;

  {Determine the parent to die, according to the probabilistic criterion}
  fw:=0; sw:=0;
  For i:=0 to num_var do
   If (simplex[i].dum_fitness>fw) and (i<>smin) then
   begin
    sw:=i; fw:=simplex[i].dum_fitness;
   end;
  pw:=simplex[sw].fitness;

  {Store the randomized worst vertex}
  For i:=0 to num_var-1 do pstart[i]:=simplex[sw].x[i];

  {Compute the centroid of the simplex}
  For i:=0 to num_var-1 do
  begin
   sum:=0;
   For j:=0 to num_var do sum:=sum+simplex[j].x[i];
   xcent[i]:=(sum-pstart[i])/num_var;
  end;

  {Make a reflection step}
  fac:=0.5+random;
  NewInd(num_var, num_obj, eval, 1+fac, -fac, F, xcent, pstart, x_lower, x_upper,
low_bound, up_bound, var_rigid, BoundReached, tempind);
  IndToGrid(pop_size, num_obj, division, tempind, pop, fmin, fmax);
  CalcIndFitness(pop_size, num_obj, minmax, f_limit, tempind, pop);
  tempind.dum_fitness:=tempind.fitness+random*temperature;
  pref:=tempind.fitness;
  fref:=tempind.dum_fitness;

  {*** Case 1: The reflection point is both non-dominated and feasible, and
replaces the worst vertex ***}
  If (tempind.feasible) and (tempind.non_domin) and (pref<pmax) then
  begin
   temperature:=ratio*temperature;
   lpos:=ipos[smax];
   CopyNewInd(num_var, num_obj, pop[lpos], tempind);
   If lpos=0 then ZeroReplaced:=True;
  end
  else
```

```
  {*** Case 2: The reflection point, albeit dominated, is better than the worst
vertex;
     hence the simplex is expanding towards the direction of reflection ***}
  If pref<pw then
  begin
   {Accept the reflection point and reduce the temperature if the new point is both
non-dominated and feasible}
   If (tempind.feasible) and (tempind.non_domin) then
temperature:=ratio*temperature;
   lpos:=ipos[sw];
   CopyNewInd(num_var, num_obj, pop[lpos], tempind);
   If lpos=0 then ZeroReplaced:=True;

   {Continue towards a line minimisation direction, by employing subsequent random
expansion steps, until a non-dominated individual is located; otherwise, employ
outside contraction between xc and xr}
   If (pref<pmin) and (not BoundReached) then
   Repeat
    fac:=fac+0.5+random;
    NewInd(num_var, num_obj, eval, 1+fac, -fac, F, xcent, pstart, x_lower, x_upper,
low_bound, up_bound, var_rigid, BoundReached, tempind);
    IndToGrid(pop_size, num_obj, division, tempind, pop, fmin, fmax);
    CalcIndFitness(pop_size, num_obj, minmax, f_limit, tempind, pop);
    pexp:=tempind.fitness;

    If pexp<pref then
    begin
     temperature:=ratio*temperature;
     pref:=pexp;
     CopyNewInd(num_var, num_obj, pop[lpos], tempind);
    end
    else Break;
   until ((tempind.feasible) and (tempind.non_domin)) or BoundReached
   else
   {Outside contraction}
   begin
    fac:=random*fac;
    NewInd(num_var, num_obj, eval, 1+fac, -fac, F, xcent, pstart, x_lower, x_upper,
low_bound, up_bound, var_rigid, BoundReached, tempind);
    IndToGrid(pop_size, num_obj, division, tempind, pop, fmin, fmax);
    CalcIndFitness(pop_size, num_obj, minmax, f_limit, tempind, pop);
    pcontr:=tempind.fitness;

    If pcontr<pref then
    begin
     temperature:=ratio*temperature;
     CopyNewInd(num_var, num_obj, pop[lpos], tempind);
     If lpos=0 then ZeroReplaced:=True;
    end;
   end; {outside contraction step}

  end
  {*** Case 3: Reflection is not successful, the simplex volume decreases ***}
  else if random>p_mutation then
  begin
   {Don't accept the reflection and try an inside contraction}
   fac:=random;
```

```
   NewInd(num_var, num_obj, eval, fac, 1-fac, F, xcent, pstart, x_lower, x_upper,
low_bound, up_bound, var_rigid, BoundReached, tempind);
   IndToGrid(pop_size, num_obj, division, tempind, pop, fmin, fmax);
   CalcIndFitness(pop_size, num_obj, minmax, f_limit, tempind, pop);
   pcontr:=tempind.fitness;

   If pcontr<pw then {accept contraction}
   begin
    temperature:=ratio*temperature;
    lpos:=ipos[sw];
    CopyNewInd(num_var, num_obj, pop[lpos], tempind);
    If lpos=0 then ZeroReplaced:=True;
   end
   else {generate a random point accross the best-forward direction}
   begin
    fac:=0.5+0.5*random;
    For j:=0 to num_var-1 do pstart[j]:=simplex[smin].x[j];
    NewInd(num_var, num_obj, eval, 1+fac, -fac, F, pstart, xcent, x_lower, x_upper,
low_bound, up_bound, var_rigid, BoundReached, tempind);
    lpos:=ipos[sw];
    CopyNewInd(num_var, num_obj, pop[lpos], tempind);
    pcontr:=tempind.fitness;
    If pcontr<pw then temperature:=ratio*temperature;
    If lpos=0 then ZeroReplaced:=True;
   end;
  end
  else
  {*** Case 4: Reflection is not successful and a random individual is generated
via mutation. The algorithm selects the "long" or "short" perturbation operator,
with 50% probability ***}
  begin
   lpos:=ipos[sw];

   If random<0.5 then
   begin {generate a random point out of the usual range of the population [m-s,
m+s]}
    For j:=0 to num_var-1 do
     if random<0.5 then ptry[j]:=xmean[j]+xstdev[j]*(1+nrnd(0,1))
            else ptry[j]:=xmean[j]-xstdev[j]*(1+nrnd(0,1));
   end
   else
   begin {generate a random point in the vincinity of the worst vertex}
    For j:=0 to num_var-1 do
    begin
     if random<0.5 then fac:=(x_upper[j]-pstart[j])/3 else fac:=-(pstart[j]-
x_lower[j])/3;
     ptry[j]:=pstart[j]+fac*nrnd(0,1);
    end;
   end;

   NewInd(num_var, num_obj, eval, 1, 0, F, ptry, ptry, x_lower, x_upper, low_bound,
up_bound, var_rigid, BoundReached, tempind);
   IndToGrid(pop_size, num_obj, division, tempind, pop, fmin, fmax);
   CalcIndFitness(pop_size, num_obj, minmax, f_limit, tempind, pop);
   pmut:=tempind.fitness;
   CopyNewInd(num_var, num_obj, pop[lpos], tempind);
   If pmut<pw then temperature:=ratio*temperature;
   If lpos=0 then ZeroReplaced:=True;
```

```
  end;

  {Update propeties}
  CalcStatistics(pop_size, num_obj, num_var, fmin, fmax, imin, imax, min_fitness,
max_fitness, xmean, xstdev, pop);
  UpdateGrid(pop_size, num_obj, division, pop, fmin, fmax);
  CalcFitness(pop_size, num_obj, minmax, f_limit, dominated_id, domin_count,
feasible_count, pop);

  If temperature>(beta*(max_fitness-min_fitness)) then
    temperature:=beta*(max_fitness-min_fitness);
  If temperature<0.1 then temperature:=1; {to avoid early convergence}

  ifeasible_count:=feasible_count;

 until (eval>maxeval) and (domin_count=0) and (ZeroReplaced) and
(feasible_count=pop_size);

 For i:=0 to pop_size-1 do
 begin
  For j:=0 to num_var-1 do xopt[i,j]:=pop[i].x[j]; {Pareto set}
  For j:=0 to num_obj-1 do fopt[i,j]:=pop[i].obj[j]; {Pareto front}
 end;

 pop:=nil; dominated_id:=nil; simplex:=nil;
 ptry:=nil; xcent:=nil; xw:=nil; pstart:=nil; ipos:=nil; fmin:=nil; fmax:=nil;

end; {procedure MEAS}


procedure CalcStatistics(pop_size, num_obj, num_var: Integer; var fmin, fmax:
TArrayOfDouble; var imin, imax: Integer; var fitmin, fitmax: Double; var xmean,
xstdev: TArrayOfDouble; var pop: my_population);
var
 i, j: Integer;
 fit, sum: Double;

begin
 fitmax:=pop[0].fitness; fitmin:=fitmax;
 For i:=1 to pop_size-1 do
 begin
  fit:=pop[i].fitness;
  If fitmax<fit then fitmax:=fit;
  If fitmin>fit then fitmin:=fit;
 end;

 For j:=0 to num_obj-1 do
 begin
  fmin[j]:=pop[0].obj[j]; fmax[j]:=fmin[j];
  For i:=1 to pop_size-1 do
  begin
   fit:=pop[i].obj[j];
   If fmax[j]<fit then fmax[j]:=fit;
   If fmin[j]>fit then fmin[j]:=fit;
  end;
 end;
```

```
 For j:=0 to num_var-1 do
 begin
  sum:=0;
  For i:=0 to pop_size-1 do sum:=sum+pop[i].x[j];
  xmean[j]:=sum/pop_size;
  sum:=0;
  For i:=0 to pop_size-1 do sum:=sum+(pop[i].x[j]-xmean[j])*(pop[i].x[j]-xmean[j]);
  xstdev[j]:=SQRT(sum/(pop_size-1));
 end;

end; {procedure CalcStatistics}


procedure CalcDominMatrix(pop_size, num_obj: Integer; var minmax: TBoolArray; var
domin_matrix: T2DArrayOfDouble; var dominated_id: TIntArray; var domin_count:
Integer; var pop: my_population);
var
 i, j, k: Integer;
 fi, fj: Double;
 sign: Integer;
 inf, sup: Integer;

begin
 For i:=0 to pop_size-1 do
  For j:=i+1 to pop_size-1 do
  begin
   domin_matrix[i,j]:=0; domin_matrix[j,i]:=num_obj;
   For k:=0 to num_obj-1 do
   begin
    If minmax[k] then sign:=1 else sign:=-1;
    fi:=sign*pop[i].obj[k]; fj:=sign*pop[j].obj[k];
    If fi<fj then
    begin
     domin_matrix[i,j]:=domin_matrix[i,j]+1;
     domin_matrix[j,i]:=domin_matrix[j,i]-1;
    end
    else if fi=fj then
    begin
     domin_matrix[i,j]:=domin_matrix[i,j];
    end;
   end;
  end;

 For i:=0 to pop_size-1 do
 begin
  inf:=0; sup:=0;
  For j:=0 to pop_size-1 do
  begin
   If i=j then Continue;
   If domin_matrix[i,j]=num_obj then inf:=inf+1 else
   if domin_matrix[i,j]=0 then sup:=sup+1;
  end;
  pop[i].inferior_count:=inf; pop[i].superior_count:=sup;
 end;

 domin_count:=0;
 For i:=0 to pop_size-1 do
  If pop[i].superior_count>0 then
```

```
  begin
   dominated_id[domin_count]:=i;
   domin_count:=domin_count+1;
  end;

end; {procedure CalcDominMatrix}


procedure UpdateGrid(pop_size, num_obj, division: Integer; var pop: my_population;
var fmin, fmax: TArrayOfDouble);
var
 i, j, k: Integer;
 f0: Double;

begin

 For i:=0 to pop_size-1 do pop[i].box:=1;

 For i:=0 to pop_size-1 do
 begin
  k:=1;
  For j:=0 to num_obj-1 do
  begin
   If fmax[j]-fmin[j]>0 then f0:=(pop[i].obj[j]-fmin[j])/(fmax[j]-fmin[j])
              else f0:=0;
   f0:=f0*division;
   f0:=int(f0)+1;
   k:=j*division+min(round(f0), division);
   pop[i].box:=pop[i].box*k;
  end;
 end;

 For i:=0 to pop_size-1 do
 begin
  k:=0;
  For j:=0 to pop_size-1 do if pop[i].box=pop[j].box then k:=k+1;
  pop[i].density:=(k-1)/pop_size;
 end;

end; {procedure UpdateGrid}


procedure IndToGrid(pop_size, num_obj, division: Integer; var ind: my_individual;
var pop: my_population; var fmin, fmax: TArrayOfDouble);
var
 j, k: Integer;
 f0: Double;

begin

 k:=1; ind.box:=1;
 For j:=0 to num_obj-1 do
 begin
  If fmax[j]-fmin[j]>0 then f0:=(ind.obj[j]-fmin[j])/(fmax[j]-fmin[j])
             else f0:=0;
  f0:=f0*division;
  f0:=int(f0)+1;
  k:=j*division+min(round(f0), division);
```

```
  ind.box:=ind.box*k;
 end;

 k:=0;
 For j:=0 to pop_size-1 do if ind.box=pop[j].box then k:=k+1;
 ind.density:=k/pop_size;

end; {procedure IndToGrid}


procedure CalcFitness(pop_size, num_obj: Integer; var minmax: TBoolArray; var
f_limit: TArrayOfDouble; var dominated_id: TIntArray; var domin_count,
feasible_count: Integer; var pop: my_population);
var
 i, j: Integer;
 count: Integer;
 sum, sum1, sign, f, f0: Double;
 domin_matrix: T2DArrayOfDouble;

begin

 SetLength(domin_matrix, pop_size, pop_size);

 CalcDominMatrix(pop_size, num_obj, minmax, domin_matrix, dominated_id,
domin_count, pop);

 For i:=0 to pop_size-1 do
 begin
  sum:=0; sum1:=0; count:=0;
  For j:=0 to pop_size-1 do
  begin
   If i=j then Continue;
   If domin_matrix[i,j]=0 then sum:=sum+pop[j].inferior_count else
   if domin_matrix[i,j]<num_obj then
   begin
    sum1:=sum1+domin_matrix[j,i]/num_obj;
    count:=count+1;
   end;
  end;
  If count>0 then sum:=sum+sum1/count;
  pop[i].rank:=sum;
  pop[i].fitness:=pop[i].rank+pop[i].density;
  pop[i].non_domin:=(sum<1);
 end;

 maxfitness:=0;
 For i:=0 to pop_size-1 do if pop[i].fitness>maxfitness then
maxfitness:=pop[i].fitness;

 feasible_count:=pop_size;
 For i:=0 to pop_size-1 do
 begin
  sum:=0; count:=0;
  For j:=0 to num_obj-1 do
  begin
   If minmax[j] then sign:=1 else sign:=-1;
   f:=sign*pop[i].obj[j]; f0:=sign*f_limit[j];
   If f>f0 then
```

```
   begin
    count:=count+1;
    sum:=sum+SQR(f-f0);
   end;
  end;
  If count>0 then
  begin
   pop[i].fitness:=maxfitness+pop[i].fitness+sum;
   feasible_count:=feasible_count-1;
   pop[i].feasible:=False;
  end;
 end;

 domin_matrix:=nil;

end; {procedure CalcFitness}


procedure CalcIndFitness(pop_size, num_obj: Integer; var minmax: TBoolArray; var
f_limit: TArrayOfDouble; var ind: my_individual; var pop: my_population);
var
 j, k: Integer;
 find, fj: Double;
 sign: Integer;
 inf, sup, count: Integer;
 domin_array: TArrayOfDouble;
 sum, sum1, sign1, f, f0: Double;

begin

 SetLength(domin_array, pop_size);

 For j:=0 to pop_size-1 do
 begin
  domin_array[j]:=0;
  For k:=0 to num_obj-1 do
  begin
   If minmax[k] then sign:=1 else sign:=-1;
   find:=sign*ind.obj[k]; fj:=sign*pop[j].obj[k];
   If find-fj<0 then domin_array[j]:=domin_array[j]+1 else
  end;
 end;

 inf:=0; sup:=0;
 For j:=0 to pop_size-1 do
 begin
  If domin_array[j]=num_obj then inf:=inf+1 else
  if domin_array[j]=0 then sup:=sup+1;
 end;
 ind.inferior_count:=inf; ind.superior_count:=sup;

 sum:=0; sum1:=0; count:=0;
 For j:=0 to pop_size-1 do
 begin
  If domin_array[j]=0 then sum:=sum+pop[j].inferior_count+1 else
  if domin_array[j]<num_obj then
  begin
   sum1:=sum1+1-domin_array[j]/num_obj;
```

```
   count:=count+1;
  end;
 end;
 If count>0 then sum:=sum+sum1/count;
 ind.rank:=sum;
 ind.fitness:=ind.rank+ind.density;
 ind.non_domin:=(sum<1);

 sum:=0; count:=0; ind.feasible:=True;
 For j:=0 to num_obj-1 do
 begin
  If minmax[j] then sign1:=1 else sign1:=-1;
  f:=sign1*ind.obj[j]; f0:=sign1*f_limit[j];
  If f>f0 then
  begin
   count:=count+1;
   sum:=sum+SQR(f-f0);
  end;
 end;
 If count>0 then
 begin
  ind.fitness:=ind.fitness+maxfitness+sum;
  ind.feasible:=False;
 end;

 domin_array:=nil;

end; {procedure CalcIndFitness}

procedure NewInd(num_var, num_obj: Integer; var eval: Integer; a1, a2: Double; F:
TVectorMathFuncMult; var x1, x2, xmin, xmax, low_bound, up_bound: TArrayOfDouble;
var var_rigid: TBoolArray; var boundary: Boolean; var ind: my_individual);
{Generates a feasible point according to the formula x[i] = a1*x1[i] + a2*x2[i]}
var
 i: Integer;
 callF: Boolean;

begin
 boundary:=False;

 With ind do
 begin
  For i:=0 to num_var-1 do
  begin
   x[i]:=a1*x1[i]+a2*x2[i];
   If var_rigid[i] then {feasible space = user specified bounds}
   begin
    If x[i]<xmin[i] then
    begin
     x[i]:=xmin[i]; boundary:=True;
    end
    else if x[i]>xmax[i] then
    begin
     x[i]:=xmax[i]; boundary:=True;
    end;
   end
   else
   begin {feasible space = physical bounds}
```

```
    If x[i]<low_bound[i] then
    begin
     x[i]:=low_bound[i]; boundary:=True;
    end
    else if x[i]>up_bound[i] then
    begin
     x[i]:=up_bound[i]; boundary:=True;
    end;
   end;
  end;
  callF:=F(x, obj);
  rank:=0; density:=0; fitness:=0; inferior_count:=0; superior_count:=0;
min_xdist:=0; min_fdist:=0;
 end;
 eval:=eval+1;

end; {procedure NewInd}


procedure CopyNewInd(num_var, num_obj: Integer; var oldind, newind: my_individual);
var
 i: Integer;
begin
 With oldind do
 begin
  For i:=0 to num_var-1 do x[i]:=newind.x[i];
  For i:=0 to num_obj-1 do obj[i]:=newind.obj[i];
  rank:=newind.rank;
  density:=newind.density;
  fitness:=newind.fitness;
  inferior_count:=newind.inferior_count;
  superior_count:=newind.superior_count;
 end;
end; {procedure CopyNewInd}


function nrnd(mi, sigma: Double): Double;
var
 z, f, nrnd1: Double;
begin
 z := sqrt(-2*ln(random));
 f := 2 * pi * random;
 nrnd1 := z * cos(f);
 nrnd := sigma * nrnd1 + mi;

end; {function nrnd}
```