Iterative optimization RCO: A "Ruler & Compass" deterministic method

Maurice Clerc

Iterative optimization — RCO: A “Ruler & Compass” deterministic method

Maurice Clerc*

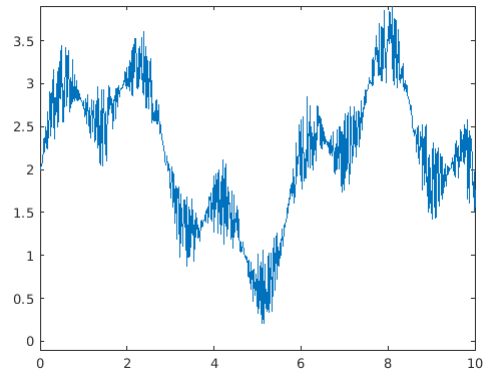
2024-04-01

1 Introduction

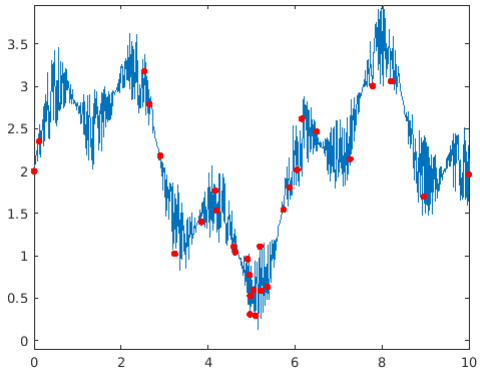
Take a piece of paper and a pen, and draw any "reasonable" function (i.e., not entirely random), even one that varies greatly, as shown in the figure 1a. For many algorithms, finding the minimum would be challenging, but the Ruler & Compass Optimizer (RCO) described here requires very few iterations. Furthermore, it is deterministic. How is this possible?

No magic, just elementary geometry.

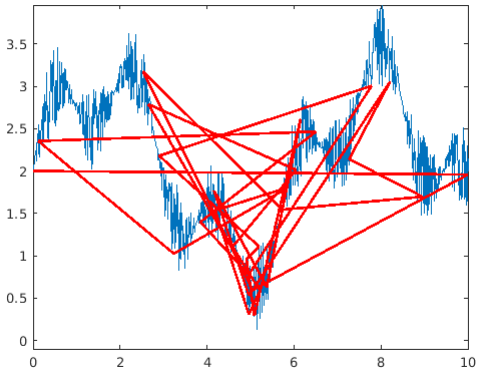
*Maurice.Clerc@WriteMe.com



(a) Landscape



(b) 32 constructed points



(c) Path followed

Figure 1: RCO on a test function.

2 Principle

The iterative construction of positions to sample in the search space can be detailed on a very simple example, defined by

$$f(x) = (x-3)^2 \quad for\, x \in [0, 10]$$

The only parameter is an estimated lower bound. Here -5 to better see what happens, although -0.1 would be sufficient.

Tables 1 to 5 illustrate the process. Of course, in higher dimension D lines become (hyper-)planes but the method is the same. At each time step we consider D planes defined by $D+1$ points and their intersection point with the lower bound plane. There are two cases:

- If its position lies outside the search space (possibly at infinity), then the new position is determined by a weighted combination of the 2^D previous positions.
- If its position is within the search space, then it is kept as the new position.

Table 1: Principle. Step 1

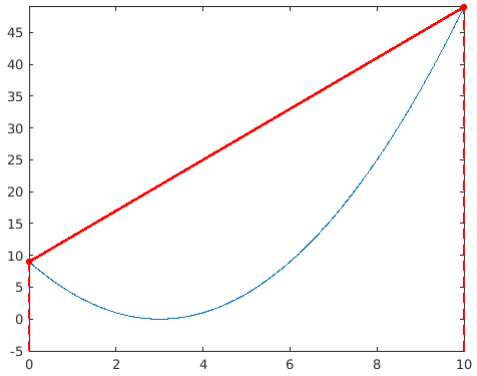
An horizontal lower bound is defined. The "corners" are evaluated. The line connecting them intersects the lower bound beyond the search space, this intersection point is rejected and will be replaced with another one (step 2).


Table 2: Principle. Step 2

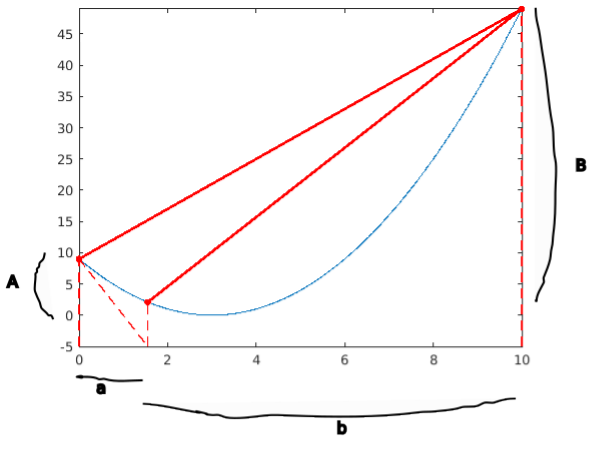
The new position is defined so that $a/b = A/B$. In dimension 1 it can be done thanks to a "Ruler & Compass" construction. Then the position is evaluated to define the third point.


Table 3: Principle. Step 3

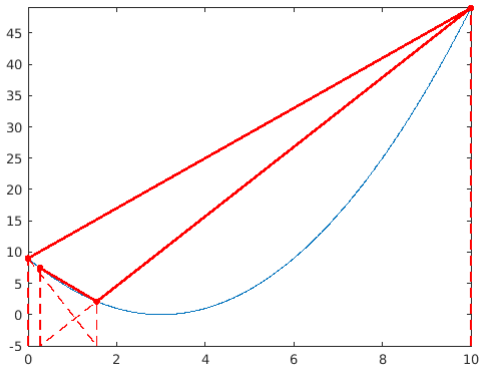
Now the line joining the two last points does intersect the lower bound on a point whose position is inside the search space. This position is then maintained and evaluated. The same process is repeated, again ...
 aine

Table 4: Principle. Step 9, zoom

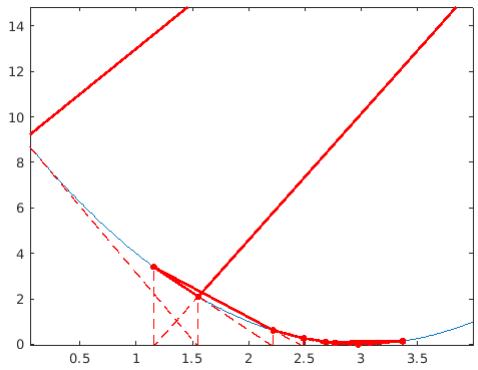
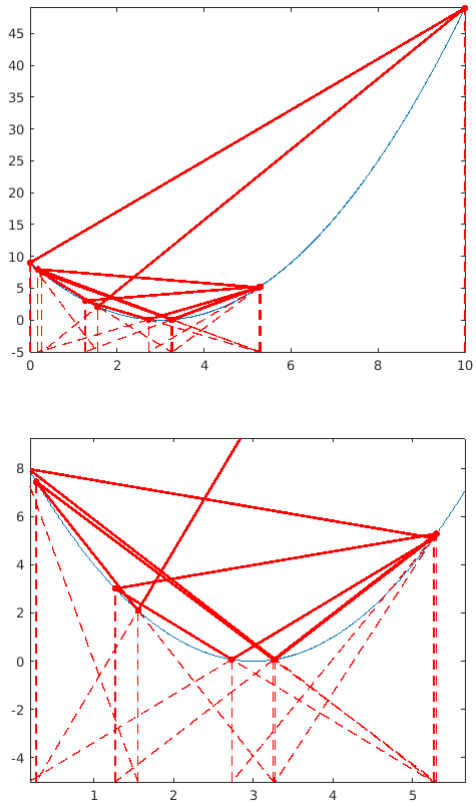
... and again.


Table 5: Principle. Step 10, and zoom

After Step 9, the line connecting the last two points is nearly horizontal and intersects the lower boundary outside of the search space. Consequently, we generate a new position using the same method as in Step 2. This occurrence becomes increasingly frequent as the positions approach the solution. Hence, the new position often approximates the mean of the last two positions, which is a favorable behavior for better approaching the solution.


3 Examples

The definitions of the problems are given in the source codes (8.4.1). Some classical problems have been shifted in order to be not too easy.

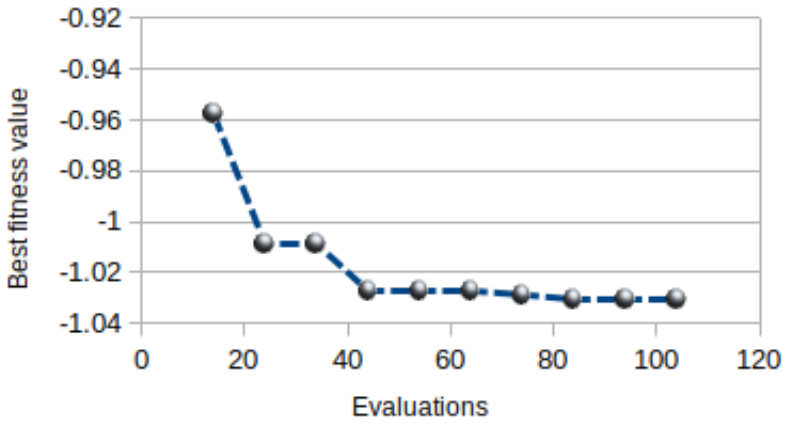
3.1 Six Hump Camel Back

The dimension of the problem is two, so the landscape has four corners. The search space is $[-2, 2] \times [-1, 1]$.

The minimum is -1.031628453489877. A sure lower bound is -1.1.

Table 6: Six Hump Camel Back

Constructed points (evaluations)	Best fitness
14	-0.957541
24	-1.008877
34	-1.008877
44	-1.027089
54	-1.027089
64	-1.027089
74	-1.028805
84	-1.030535
94	-1.030535
104	-1.030535



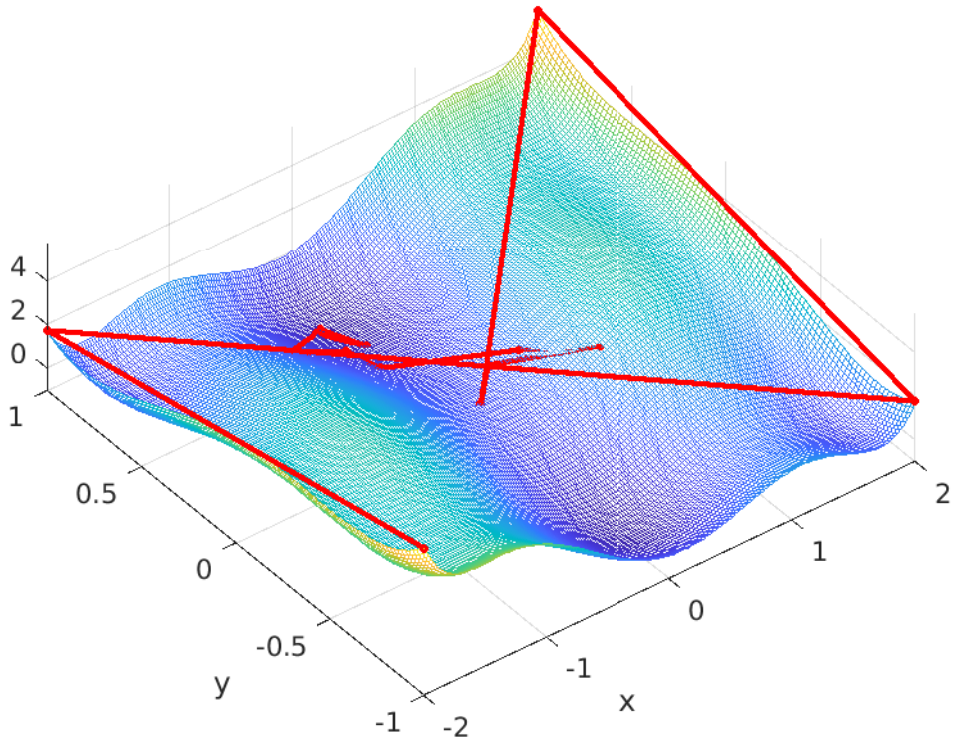


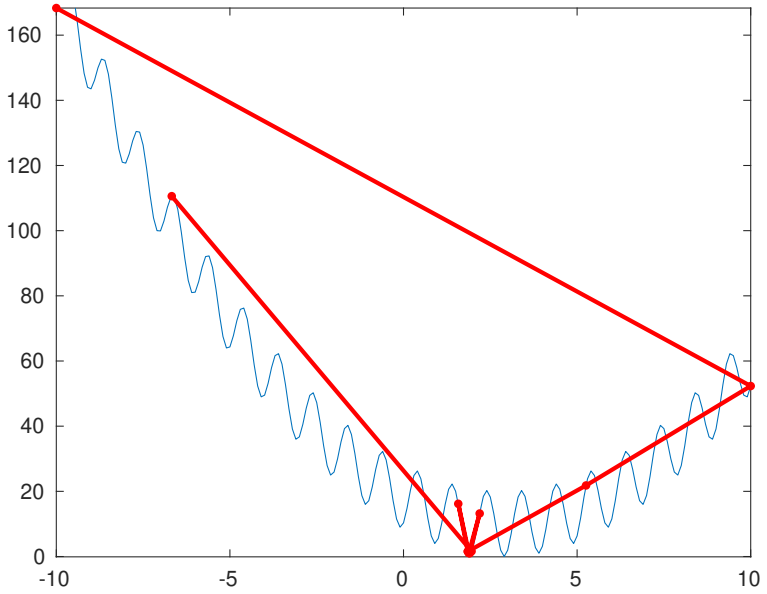
Figure 2: Six Hump Camel Back. Lower bound -1.1. After the construction of 4+10 points the result is -0.9575.

3.2 Shifted Rastrigin

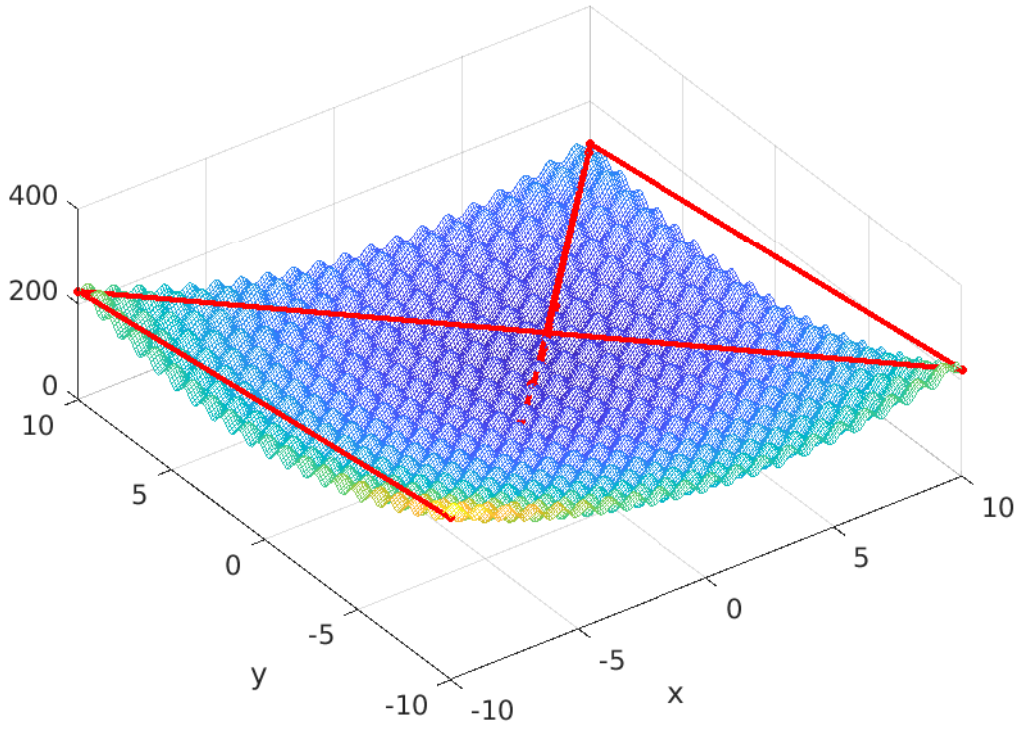
The minimum is zero so a sure lower bound is -0.1.

Table 7: Shifted Rastrigin

Dimension	Constructed points (evaluations)	Best fitness
1	52	0.000557
2	1004	0.001511
3	1508	0.002548
4	50016	0.001735
5	250032	0.004518



(a) 1D, 12 points



(b) 2D, 14 points

Figure 3: Shifted Rastrigin

3.3 Rosenbrock

The minimum is zero so a sure lower bound is -0.1.

Table 8: Rosenbrock

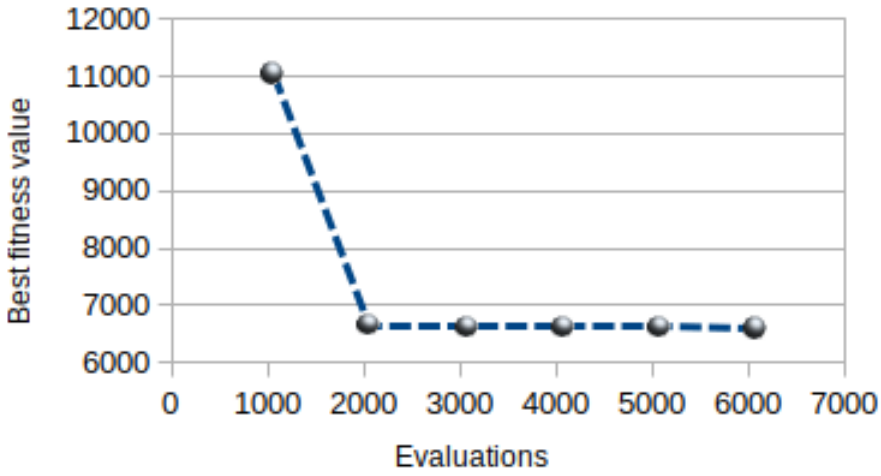
Dimension	Evaluations	Best fitness
2	252	0.002788
3	1508	0.001003
4	10016	0.003756
5	15032	0.005719

3.4 Pressure Vessel

There are four values to find, the two first ones are discrete. The minimum value is 6059.714335048436, as proved in Yang et al. 2013.

The lower bound used here is 6000. Due to the discrete variables, the algorithm encounters difficulties in converging.

Table 9: Pressure Vessel



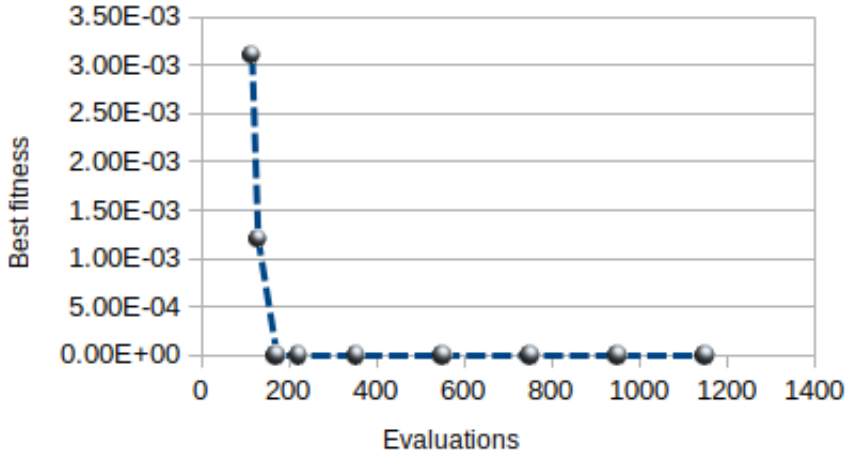
Evaluations	Best fitness
1052	11056.81
2049	6646.71
3074	6615.977
4074	6615.977
5074	6615.977
6076	6592.011

3.5 Gear Train

There are four values to find, all discrete. The minimum is 2.700857×10^{-12}.

Here the lower bound is simply set to 0. The algorithm quickly proposes a solution that is far from optimal, and then stagnates without further improvement.

Table 10: Gear Train. Discretization is applied only at the very end to all values, so more search effort does not always imply a better result.



Evaluations	Best fitness
116	3.10E-03
130	1.20E-03
173	6.65E-09
223	6.65E-09
1753	6.65E-09

4 Comparison

Let's compare RCO with a slightly improved version of Standard PSO (SPSO-Dicho on my technical website (Clerc 2024b). To slightly favor PSO, I've selected the best run out of five. The results are presented in 11. Also, refer to the figure 4 for Shifted Griewank, where the two methods appear to be equivalent, particularly for dimensions 1 to 5 and the number of evaluations respectively set to 100, 200, 5000, 10000, and 20000.

In lower dimensions (less than 10), RCO performs better and sometimes significantly so. However, for higher dimensions, RCO stagnates while PSO continues to find better solutions, as evidenced by Shifted Rastrigin and Rosenbrock.

Table 11: RCO vs PSO

Problem	Dimension	Evaluations	RCO	PSO
Six Hump Camel Back	4	54	**-1.031277**	-1.011030
		504	**-1.031473**	-1.031628
Shifted Rastrigin	1	52	**0.000557**	0.510724
	2	1004	**0.001511**	0.4397499
	5	250032	**0.004578**	0.994959
	10	251024	35.246	**1.9899**
Rosenbrock	2	252	**0.002788**	266.94
	5	15032	**0.005719**	0.0864
	10	101024	8.996	**5.19e-08**
Pressure Vessel	4	3074	**6615.977**	6890.347
		5074	**6615.977**	6821.931
		20074	**6583.75**	6820.410
Gear Train	4	173	6.65e-09	**2.35e-09**
		1154	6.65e-09	**2.35e-09**
		199752	1.18e-09	**1.26e-09**

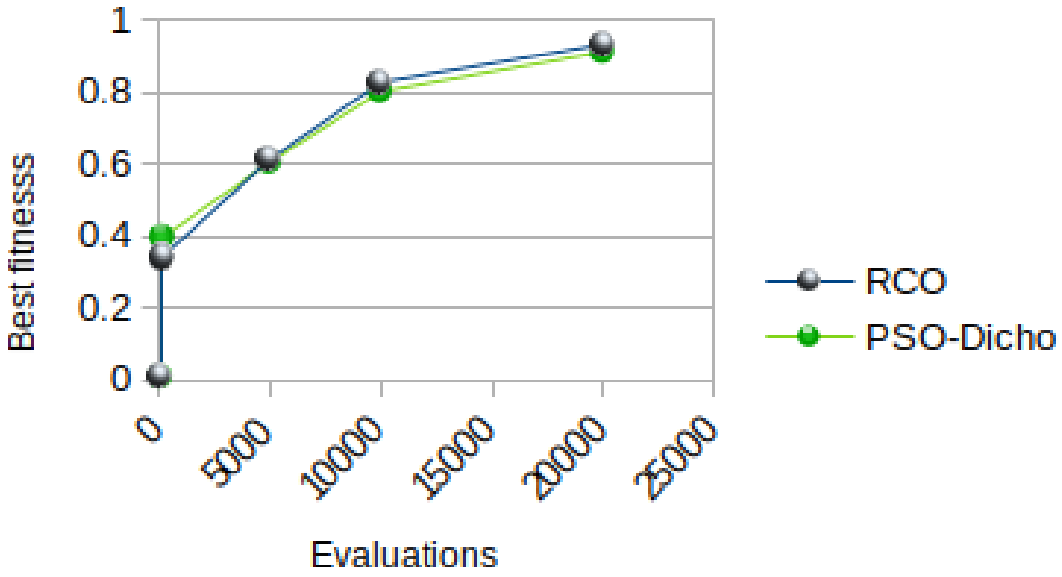


Figure 4: Shifted Griewank. RCO and PSO are equivalent.

5 Complexities

To evaluate an algorithm a common practice is to perform a theoretical analysis of its space complexity and of its time complexity. Note that, however, and as explained in Clerc 2024a, this is not always pertinent and a safer approach is to estimate the real memory space that is used and the real computing time.

Here, at each time step, there are two possible cases:

1. solving a linear system of D equations to find a new position with D coordinates. The needed space is $D^2 + D$ and the time complexity $O(D^3)$.
2. computing a linear combination of 2^D positions with D coordinates. The needed space is $D \times 2^D$ and the number of multiplications is2^D.

To assess an algorithm, it's common practice to conduct a theoretical analysis of both its space complexity and time complexity. However, as explained in Clerc 2024a, this approach isn't always appropriate, and a more reliable method is to estimate the actual memory usage and computational time.

In this context, there are two possible scenarios at each time step:

1. Solving a linear system of D equations to determine a new position with D coordinates. The required space is $D^2 + D$, with a time complexity of $O(D^3)$.
2. Computing a linear combination of 2^D positions, each with D coordinates. The required space is $O(D \times 2^D)$, and the number of multiplications is $O(2^D)$.

However such a classical reasoning does not correspond to the behavior of the algorithm in practice. The point is that the second situation usually does not occur very often. Moreover it depends on the landscape of the problem and also on the "budget" (the number of evaluations).

Let's consider, for example, the computing time per evaluation for two problems: Planes and Shifted Rastrigin. We have already seen the Shifted Rastrigin and Planes is defined by

$$f(x) = \sum_{d=1}^{D} (x(d) - 3) \tag{1}$$

.The figure 5 depicts its landscape in two dimensions.

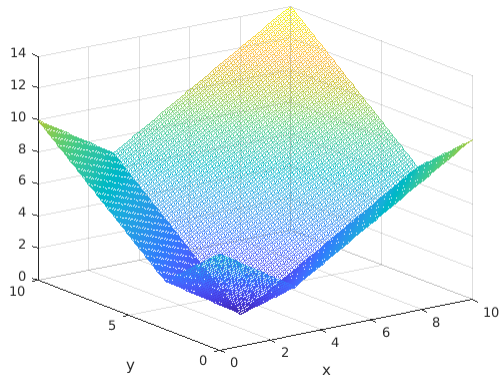
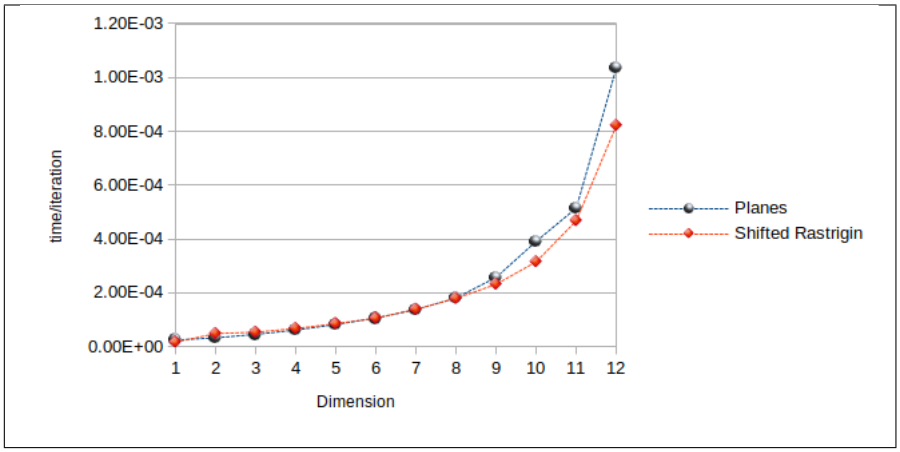


Figure 5: 2D Planes problem landscape.

As we can see on table 12 the real computing time indeed grows exponentially, but more like 1.35^D than like 2^D.

Table 12: Computing time per iteration

Dimension	Planes	Shifted Rastrigin
1	2.40E-05	1.60E-05
2	3.20E-05	4.80E-05
3	4.50E-05	5.30E-05
4	6.09E-05	6.69E-05
5	8.17E-05	8.47E-05
6	1.05E-04	1.05E-04
7	1.37E-04	1.38E-04
8	1.80E-04	1.78E-04
9	2.57E-04	2.31E-04
10	3.91E-04	3.15E-04
11	5.15E-04	4.67E-04
12	1.04E-03	1.09E-03



6 When it doesn't work

On some problems, even continous and of low dimension, RCO doesn't work well. Let's consider for example the Frequency-Modulated Sound Waves (FM) from the CEC 2011 competition benchmark (Das and Suganthan 2011). The search space is $[-6.4, 6.35]^6$ and the minimum is zero. Even after 50064 evaluations the best final value is 24.07. This is because many intersection positions (21238) are outside the search space (i.e. inacceptable).

Note that in such case it may be a little bit beneficial to expand the search space, if possible. If it is now $[-500, 500]^6$ there are 17322 inacceptable intersections and the final best value is 17.42. But many classical stochastic algorithms find a far better solution, precisely because the stochasticity can cope with the highlly cahotic landscape (see the cross section 6).

In some instances, even when problems are continuous and of low dimension, RCO doesn't perform well. For instance, let's examine the Frequency-Modulated Sound Waves from the CEC 2011 competition benchmark (Das and Suganthan 2011). The search space is $[-6.4, 6.35]^6$ and the

minimum value is zero. Despite 50064 evaluations, the best final value achieved is 24.07. This is primarily due to a considerable number of intersection positions (21238) lying outside the search space, rendering them unacceptable.

It's worth noting that in such cases, expanding the search space might provide some benefit if feasible. For example, if the search space is expanded to $[-500, 500]^6$, there are 17322 unacceptable intersections, resulting in a final best value of 17.42. However, many classical stochastic algorithms tend to discover significantly better solutions. This is because stochasticity can effectively navigate a highly chaotic landscape, as is the one of FM (see Figure: 6).

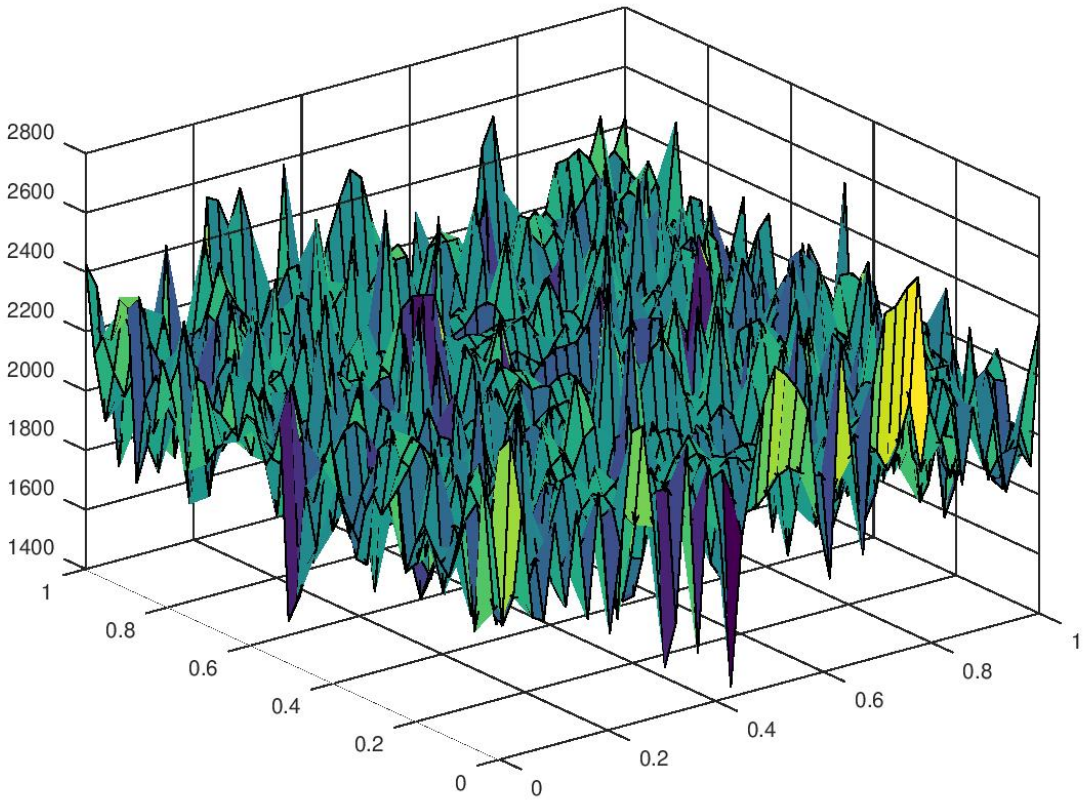


Figure 6: Frequency-Modulated Sound Waves problem. Cross section on dimensions 2 and 5.

7 Conclusion

Even in low dimensions, evaluating the objective function can sometimes be very expensive. In such cases, RCO may be useful as it may propose a sure solution after a few evaluations. However, there are some drawbacks:

- It requires a reasonably good lower bound. On the other hand, compared to most other methods, this is the only user-defined parameter. In practice, for real problems, such a bound is often known.
- It does not work well on some problems, even of low dimension.
- It does not perform well on discrete problems, particularly when all variables are discrete. However, it still seems to be usable when only some variables are discrete.
- Its computing time per iteration grows exponentially with the dimension of the search space. It does not grow as quickly as theoretically expected, but nonetheless, in practice, it can be difficult to use on a laptop for high-dimensional problems.

8 Appendix

8.1 A bit of geometry

For the "Ruler & Compass" construction in dimension 1 and as seen on the figure 2 we have to be able to define two segments **a** and **b** so that $\mathbf{a}/\mathbf{b} = \mathbf{A}/\mathbf{B}$. The method is more than 2600 years old (Thales of Miletus) so you may have forgotten it ...

The figure 7 shows how to do.

To perform the "Ruler & Compass" construction in one dimension, as depicted in Figure 2 , it is necessary to define two segments, denoted as **a** and **b**, such that $\mathbf{a}/\mathbf{b} = \mathbf{A}/\mathbf{B}$. This method, dating back over 2600 years to Thales of Miletus, may have slipped from your memory over time...

Figure 7 illustrates the procedure.

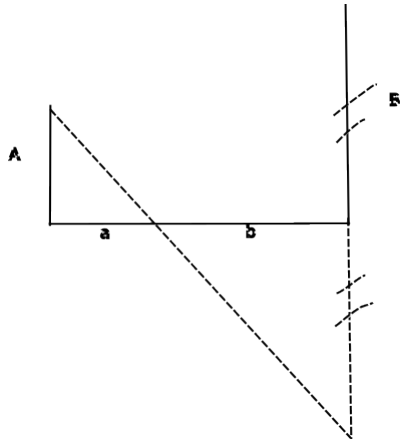
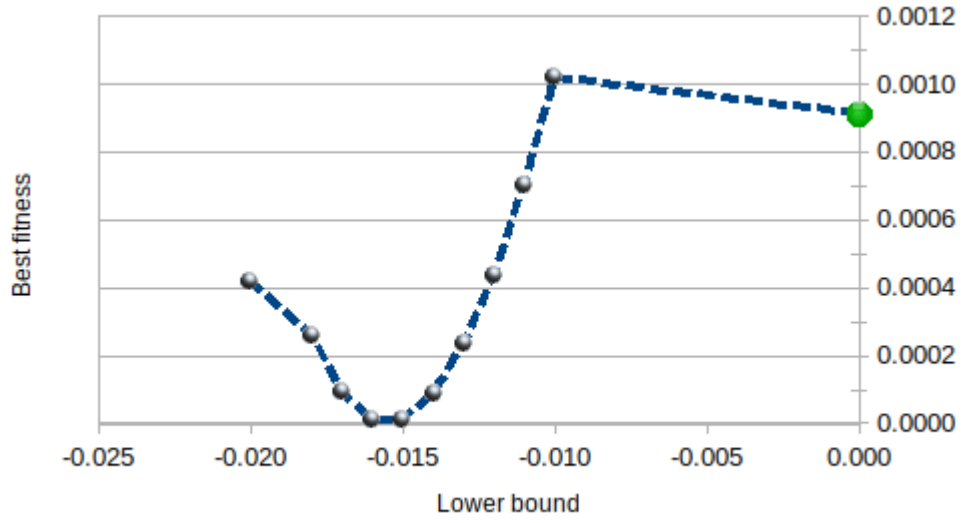


Figure 7: Ruler a&nd Compass method to define proportional segments.

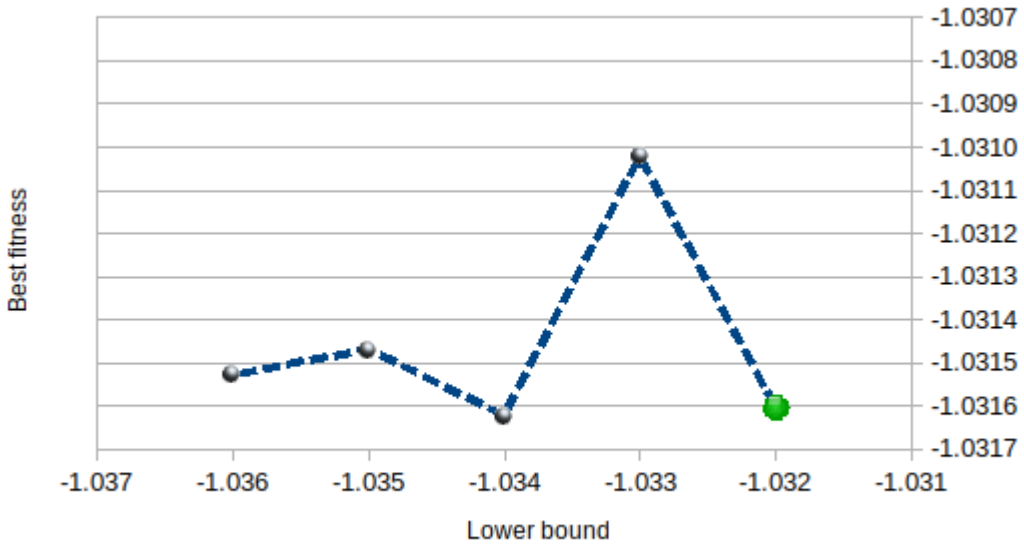
8.2 Sensitivity

There is only one user-defined parameter: the lower bound. Therefore, it is crucial to understand the algorithm's sensitivity to this parameter. As depicted in the figures 8, we observe that efficiency

is indeed sensitive to changes in the lower bound. Though the sensitivity is not extreme, it could still be beneficial to experiment with various values. It's worth noting that setting the lower bound to the exact minimum value, if known, is not the optimal choice.



(a) Parabola

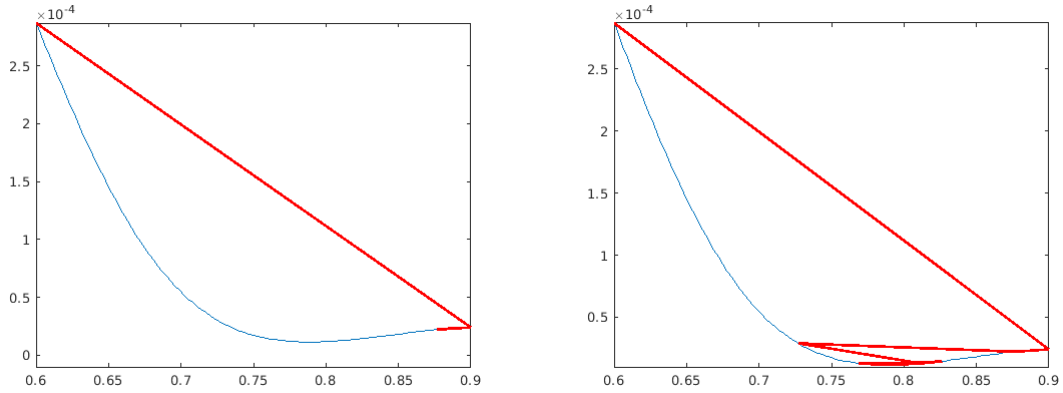


(b) Six Hump Camel Back

Figure 8: Efficiency is somewhat sensitive to the user-defined lower bound, but not excessively so.

A clear example of sensitivity can be observed when attempting to solve the optimal control problem of the bifunctional catalyst blend in the CEC 2011 competition (Das and Suganthan 2011).

As depicted in Figure 9, even a slight modification to the lower bound leads to significantly different behavior due to the very small slope. Conversely, due to the same reason, a position far from the optimal one already exhibits a value close to the minimum.



(a) Lower bound -0.00001

(b) Lower bound 0.00001

Figure 9: CEC 2011 bifunctional catalyst blend optimal control problem. 12 evaluations.

If you are unsure about the lower bound, you can utilize an adaptive one (refer to the parameter `adaptOption` in the source code). In fact, the results can occasionally be quite satisfactory, as evidenced by Table 13. Thus, it may be worthwhile to try different methods: user-defined and the adaptive ones. In the source code, there are three potential formulas for adaptation, but they are rather rudimentary. Certainly, there is room for improvement.

Table 13: Adaptive lower bound may improve the efficiency ... or the contrary!

	Dimension	Evaluations	Lower bound	Non adaptive	Adaptive 3 coeff. 0.5	Adaptive 3 coeff. 0.1
Six Hump Camel Back	2	104	-1.1	-1.031473	**-1.03157**	-0.947
Shifted Rastrigin	3	1508	-0.1	0.002548	0.06729	**3.27E-05**
	4	50016	-0.1	0.00173	**1.78E-15**	**0**
Rosenbrock	3	1508	-0.1	0.001	0.1866	0.238
	4	10016	-0.1	0.003756	**7.52E-22**	**8.94E-24**
Pressure Vessel	4	6076	6000	6592	6955	1.29E+05
Gear Train	4	1753	0	6.65E-09	1.38E-06	**6.60E-10**

8.3 More examples

For amusement, here are a few 1D and 2D figures illustrating how rapidly the algorithm approaches the solution, thanks to steps defined purely by geometry. Initially, they are quite large and gradually diminish in size as they approach the solution's position.

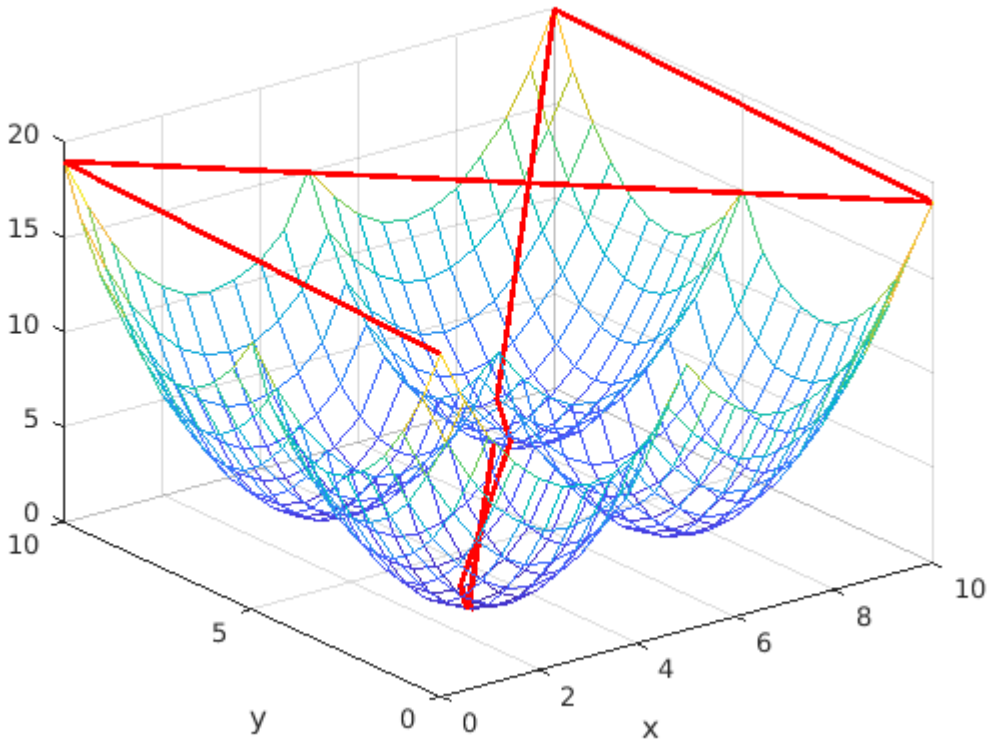
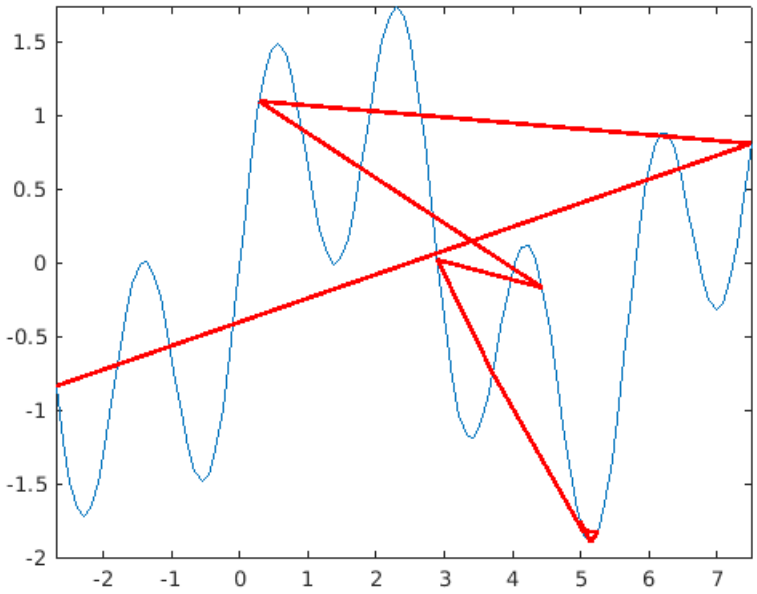
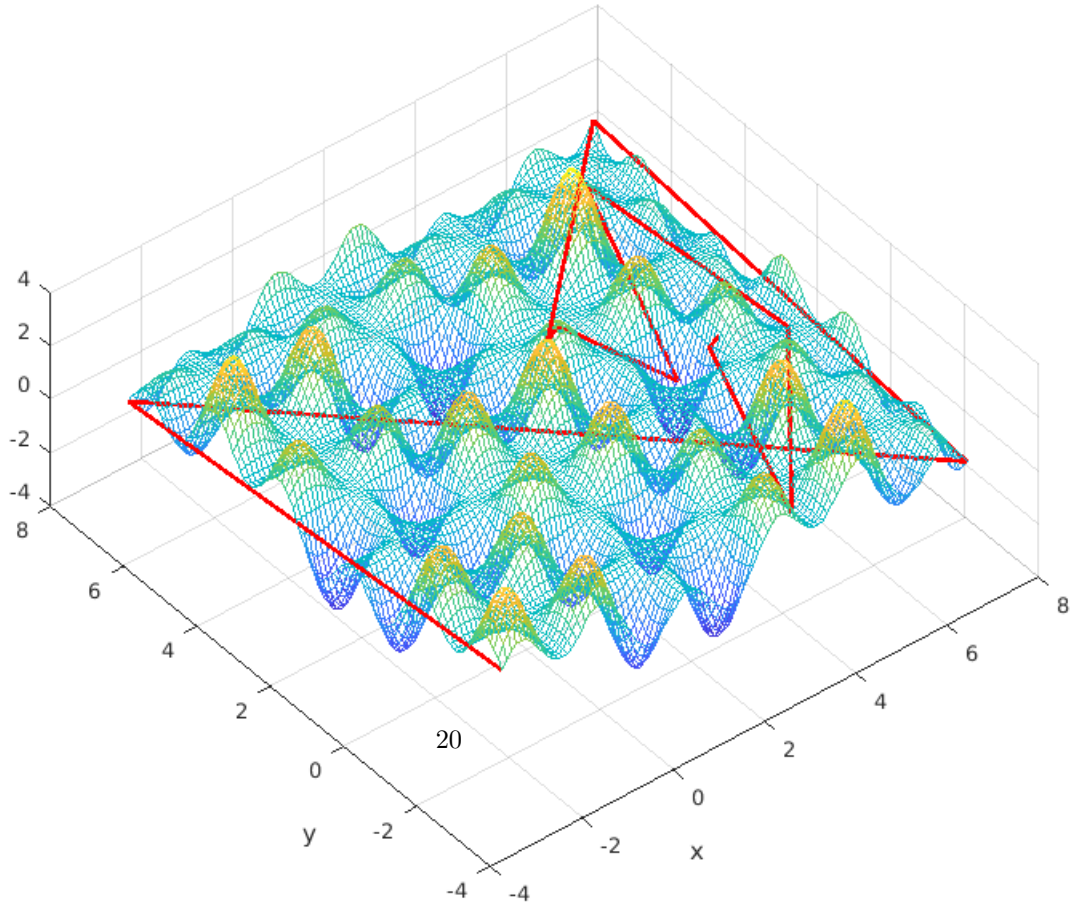


Figure 10: Multiparaboloid. 14 evaluations, final value 0.38 (real minimum 0).



(a) 12 evaluations, final value -1.899584 (real minimum -1.8996).



(b) 16 evaluations, final value -1.732.

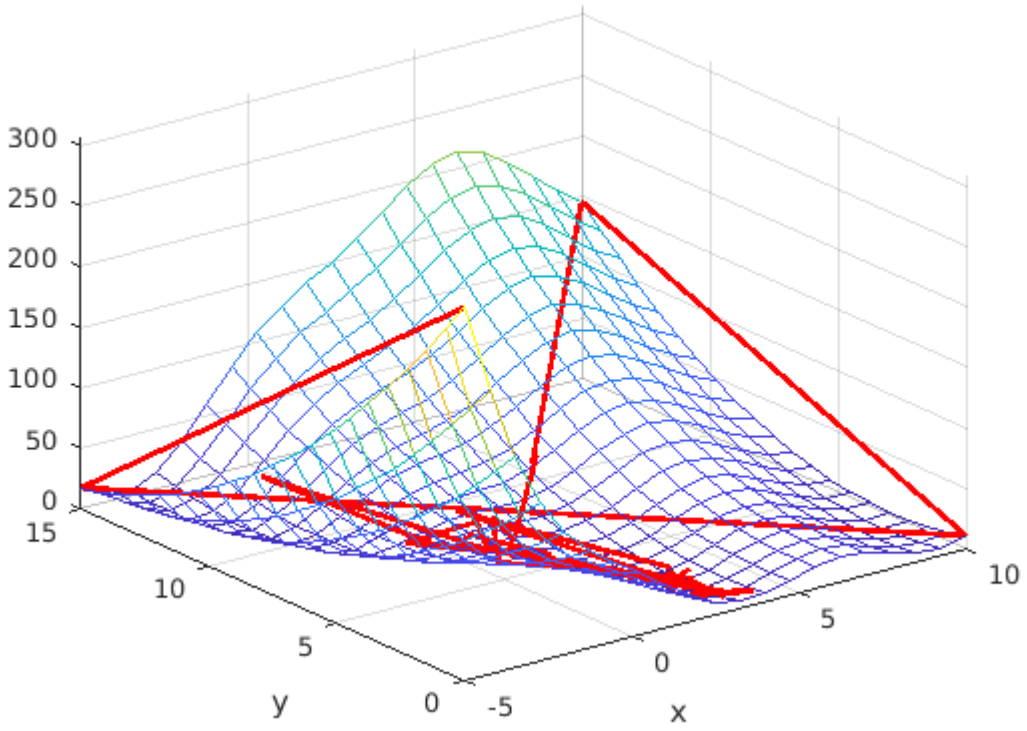


Figure 12: Branin. 44 evaluations, final value 0.3988 (real minimum 0.397887).

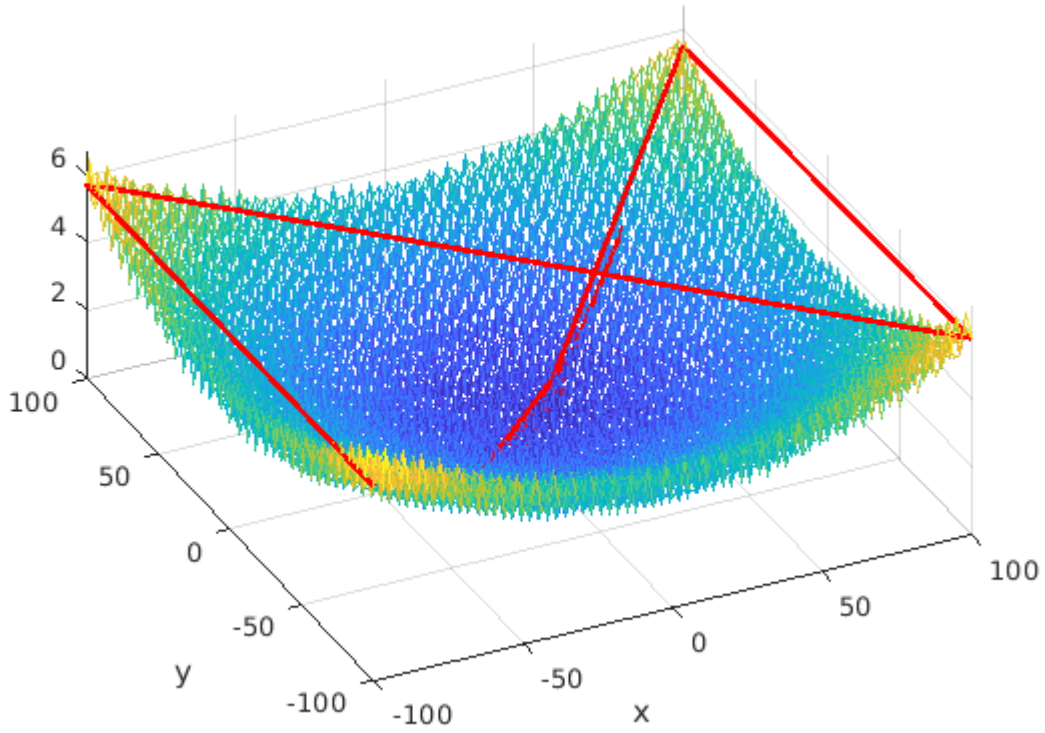


Figure 13: Shifted Griewank. 14 evaluations, final value 0.34 (real minimum 0).

8.4 Source codes

Written for Octave/Matlab©. Depending on the versions a few instructions (especially those for plotting) may not work with Octave.

Here is a typical run for Pressure Vessel:

```
>> RCO(11,4,3000)
================================================
 Best after init 4.636519e+07
 function 11, dimension 4, budget 3000
 lowerBound 6000.000000
 Evaluations 3074 (including 58 for discretisation)
 noWayCount 12
 outsideCount 341
 fBest 6.615977e+03
 on
    1.0000    0.5000   50.2674   97.8799
>>
```

8.4.1 Problems

See also in the attached cec.zip file:
`c_bifunc_data.mat`
`data6Bus.m`
`diffsolv.m`
cec2011_TNEP.m
that are needed for some problems of the CEC 2011 competition.

```
function [xMin, xMax, lowerBound,quantis]=get_func(func,D)
One=ones(1,D);
quantis=[];
switch func
  case -3 % abs(sin(x) + ax)
    xMin=zeros(1,D);
    xMax=10*One;
    lowerBound=-0.1;

  case -2 % Test
    xMin=zeros(1,D);
    xMax=10*One;
    lowerBound=-0.1;

  case {-1,0,3,4,5,8}
    xMin=zeros(1,D);
    xMax=10*One;
    lowerBound=-0.1;

  case 6
    xMax=10*One;
    xMin=-xMax;
    lowerBound=-0.1;

  case 7
    xMin=-One;
    xMax=One;
    lowerBound=-0.1;

  case 9 % Rosenbrock
    xMax=100*One;
    xMin=-xMax;
    lowerBound=-0.1;
    %lowerBound=0;
  case 10    % Six Hump Camel Back
    xMin=[-2,-1];
    xMax=-xMin;
```

```
  lowerBound=-1.1;
  %lowerBound=-1.031628453489877;

case 11 % Pressure Vessel (D=4)
  q=0.0625;
  quantis=[q,q,0,0];
  xMin=[q,q,10,10];
  xMax=[99,99,200,200];
  lowerBound=6000;
  %lowerBound=6059.714335048436;

case 12 % Gear Train (D=4)
  D=4;
  ones4=ones(1,4);
  xMin=12*ones4;
  xMax=60*ones4;
  quantis=[1,1,1,1];
  lowerBound=0;
  %lowerBound=2.700857e-12;

case 13 % Griewank
  xMax=100*One;
  xMin=-xMax;
  lowerBound=0;

case 14 % Sin Sin
  xMin=-2.7*One;
  xMax=7.5*One;
  lowerBound=-2*D;

case 15 % CEC 2011. Frequency-Modulated (FM) Sound Waves
  % COUNTER EXAMPLE
  D=6;
  xMin=-6.4*One;
  xMax=6.35*One;
  lowerBound=-0.0001;

case 16 %CEC 2011. The bifunctional catalyst blend optimal control problem
  D=1;
  xMin=0.6;
  xMax=0.9;
  lowerBound=-0.00001;

case 17 % CEC 2011. Transmission Network Expansion Planning
  % Difficult. Finds only 231 after 50000 evaluations
  % as most heuristics find the minimum 220.
```

```
    % And quite sensitive to the lower bound.
    D=7;
    xMin=One;
    xMax=15*One;
    lowerBound= 150;

    % ...
  case 39 % Branin
    D=2;
    xMin=[-5, 0];
    xMax=[10,15];
    lowerBound=0;

  otherwise
    xMin=-10*One;
    xMax=10*One;
    lowerBound=0;%-0.1;
end
end
```

```
function RCO(func,D,tMax) % Ruler and Compass Optimisation
rng(123456789) % Only for test with function -2
%{
    Maurice.Clerc@WriteMe.com
    2024-04-03

A simple deterministic optimisation method.
On dimension 1 you just need a ruler and a compass to apply it.
And also on higher dimension D ... assuming you have
a D-ruler and a D-compass!
%}
warning('off') % Don't display warnings (in particular for singular matrix)
global noWayCount outsideCount
noWayCount=0;
outsideCount=0;
qAlways=true; % For (partly) discrete problem
% false (experimental) => discretisation only at the very end, on xBest
% true => at each time step
tQuantis=0; % For (partly) discrete problems. Number of improvements
adaptLBcoeff=0.5; %0.1; % ~=0   => adaptive lower bound
                  % 0 => only user defined lower bound (see get_func.m)
adaptOption=2; % 1, 2 or 3  Adaptation method.It may be useful to
                %           try the three.
% Possible plots for D=1 or 2
plotLandscape=true;
```

```
plotPoints=false;
plotLine=true;
plotDash=false; % Only for D=1
plotRun=(plotPoints || plotLine || plotDash) && D<=2;
fprintf('\n============================================================')
% Define the problem
[xMin, xMax, lowerBound,quantis]=get_func(func,D);
tStart=cputime;
% ------------------------------- Initialisation
% Define all "corners"
x=corners(xMin,xMax);
% Evaluate them
[nPoints,~]=size(x);
for n=1:nPoints
  f(n)=fit(x(n,:),func);
end
% Save the best
[fBest,Ind]=min(f);
fprintf('\n Best after init %e',fBest)
xBest=x(Ind,:);
fBestQ=fBest;
if plotRun
  line=zeros(tMax+nPoints,D+1); % Just to speed up
  for n=1:nPoints
    line(n,1:D)=x(n,:); line(n,D+1)=f(n);
  end
end
% -------------------------------
tp=nPoints;
for t=1:tMax
  if adaptLBcoeff ~=0 % Adaptive lower bound
    if ~isempty(quantis) && qAlways
      fB=fBestQ;
    else
      fB=fBest;
    end

    switch adaptOption
      case 1 % Method 1
        if fB>=0
          LB=adaptLBcoeff*fB;
        else
          LB=(1+adaptLBcoeff)*fB;
        end

      case 2 % Method 2
```

```
     % LB=(lowerBound+fB)*adaptLBcoeff;
     LB=adaptLBcoeff*lowerBound +(1-adaptLBcoeff)*fB;

    case 3 % Method 3
      if fB>0
        LB=fB*adaptLBcoeff;
      else
       % LB=(lowerBound+fB)*adaptLBcoeff;
         LB=adaptLBcoeff*lowerBound +(1-adaptLBcoeff)*fB;
      end

  end
else % User defined lower bound
  LB=lowerBound;
end
[xn,fn]=new(x,f,xMin,xMax,func,LB,t);

if plotRun % Keep positions for future plots
  tp=tp+1;
  line(tp,1:D)=xn; line(tp,D+1)=fn;
end

if ~isempty(quantis) && qAlways
  if fn<fBestQ % Keep the best solution after discretisation
    xQ=quantify(xn,quantis);
    fQ=fit(xQ,func);
    tQuantis=tQuantis+1;
    if fQ<fBestQ
      fBestQ=fQ;
      xBest=xQ;
    end

  end
else
  if fn<fBest
    fBest=fn;
    xBest=xn;
  end
end

% Define another line (D=1) or plane (D=2) or hyperplane (D>2)
for p=1:nPoints-1
  x(p,:)=x(p+1,:);
  f(p)=f(p+1);
end
x(nPoints,:)=xn;
```

```
  f(nPoints)=fn;

end % end of the run
% ------------------------------- Results
if ~isempty(quantis)  % Some problems may be (partly) discrete
  if ~qAlways
    xBest=quantify(xBest,quantis);
    fBest=fit(xBest,func);
    tQuantis=tQuantis+1;
  else
    fBest=fBestQ;
  end
end
tEnd=cputime;
fprintf('\n function %i, dimension %i, budget %i',func,D,tMax);
if adaptLBcoeff~=0
  switch adaptOption
    case 1
      fprintf('\n Adaptation 1')
      fprintf('\n Totally adaptive lower bound, coefficient %f',adaptLBcoeff);
    case 2
      fprintf('\n Adaptation 2');
      fprintf('\n Mix user defined lower bound %e + adaptive,  coefficient %f',...
        lowerBound,adaptLBcoeff);
      fprintf('\n (no matter the sign of fBest)')
    case 3
      fprintf('\n Adaptation 3');
      fprintf('\n Mix user defined lower bound %e + adaptive,  coefficient %f',...
        lowerBound,adaptLBcoeff);
      fprintf('\n (depending on the sign of fBest)');
  end
else
  fprintf('\n lowerBound %f',lowerBound);
end
fprintf('\n Evaluations %i', tMax+nPoints+tQuantis);
if  tQuantis>0
  fprintf(' (including %i for discretisation)',tQuantis)
end
fprintf('\n noWayCount %i',noWayCount);
fprintf('\n outsideCount %i',outsideCount);
fprintf('\n fBest %e',fBest)
fprintf('\n on \n'); disp(xBest)
fprintf('\n CPU time %e \n',tEnd-tStart)
% ================================= Plots
if plotRun figure;  end
if plotLandscape && D<=2
```

```
  fPlot(func,D,xMin,xMax,lowerBound);
  hold on
end
if plotLine
  switch D
    case 1
      plot(line(:,1), line(:,2),'r-','LineWidth',2);
      %axis([xMin,xMax,lowerBound, max(line(:,2))]);
      hold on
    case 2
      plot3(line(:,1), line(:,2),line(:,3),'r-','LineWidth',2);
      hold on
  end
end
if plotPoints
  switch D
    case 1
      plot(line(:,1), line(:,2),'r.','MarkerSize',12)
      %axis([xMin,xMax,lowerBound, max(line(:,2))]);
      hold on
    case 2
      plot3(line(:,1), line(:,2),line(:,3),'r.','MarkerSize',12)
      hold on
  end
end
if plotDash && D==1
  [tMax,~]=size(line);
  for t=1:tMax-2
    X=[line(t,1),line(t+2,1)];
    Y=[line(t,2),lowerBound];
    plot(X,Y,'r--','LineWidth',1);
    hold on
  end

  for t=1:tMax % Vertical lines
    X=[line(t,1),line(t,1)];
    Y=[lowerBound,line(t,2)];
    plot(X,Y,'r--','LineWidth',1);
    hold on
  end
  %axis([xMin,xMax,lowerBound, max(line(:,2))]);

end
end
```

8.4.2 Algorithm

```
function RCO(func,D,tMax) % Ruler and Compass Optimisation
rng(123456789) % Only for test with function -2
%{
    Maurice.Clerc@WriteMe.com
    2024-04-03

A simple deterministic optimisation method.
On dimension 1 you just need a ruler and a compass to apply it.
And also on higher dimension D ... assuming you have
a D-ruler and a D-compass!
%}
warning('off') % Don't display warnings (in particular for singular matrix)
global noWayCount outsideCount
noWayCount=0;
outsideCount=0;
qAlways=true; % For (partly) discrete problem
% false (experimental) => discretisation only at the very end, on xBest
% true => at each time step
tQuantis=0; % For (partly) discrete problems. Number of improvements
adaptLBcoeff=0.5; %0.1; % ~=0   => adaptive lower bound
                  % 0 => only user defined lower bound (see get_func.m)
adaptOption=2; % 1, 2 or 3  Adaptation method.It may be useful to
               %           try the three.
% Possible plots for D=1 or 2
plotLandscape=true;
plotPoints=false;
plotLine=true;
plotDash=false; % Only for D=1
plotRun=(plotPoints || plotLine || plotDash) && D<=2;
fprintf('\n============================================================')
% Define the problem
[xMin, xMax, lowerBound,quantis]=get_func(func,D);
tStart=cputime;
% ------------------------------- Initialisation
% Define all "corners"
x=corners(xMin,xMax);
% Evaluate them
[nPoints,~]=size(x);
for n=1:nPoints
  f(n)=fit(x(n,:),func);
end
% Save the best
[fBest,Ind]=min(f);
fprintf('\n Best after init %e',fBest)
xBest=x(Ind,:);
```

```
fBestQ=fBest;
if plotRun
  line=zeros(tMax+nPoints,D+1); % Just to speed up
  for n=1:nPoints
    line(n,1:D)=x(n,:); line(n,D+1)=f(n);
  end
end
% --------------------------------
tp=nPoints;
for t=1:tMax
  if adaptLBcoeff ~=0 % Adaptive lower bound
    if ~isempty(quantis) && qAlways
      fB=fBestQ;
    else
      fB=fBest;
    end

    switch adaptOption
      case 1 % Method 1
        if fB>=0
          LB=adaptLBcoeff*fB;
        else
          LB=(1+adaptLBcoeff)*fB;
        end

      case 2 % Method 2
       % LB=(lowerBound+fB)*adaptLBcoeff;
       LB=adaptLBcoeff*lowerBound +(1-adaptLBcoeff)*fB;

      case 3 % Method 3
        if fB>0
          LB=fB*adaptLBcoeff;
        else
         % LB=(lowerBound+fB)*adaptLBcoeff;
           LB=adaptLBcoeff*lowerBound +(1-adaptLBcoeff)*fB;
        end

    end
  else % User defined lower bound
    LB=lowerBound;
  end
  [xn,fn]=new(x,f,xMin,xMax,func,LB,t);

  if plotRun % Keep positions for future plots
    tp=tp+1;
    line(tp,1:D)=xn; line(tp,D+1)=fn;
```

```
  end

  if ~isempty(quantis) && qAlways
    if fn<fBestQ % Keep the best solution after discretisation
      xQ=quantify(xn,quantis);
      fQ=fit(xQ,func);
      tQuantis=tQuantis+1;
      if fQ<fBestQ
        fBestQ=fQ;
        xBest=xQ;
      end

    end
  else
    if fn<fBest
      fBest=fn;
      xBest=xn;
    end
  end

  % Define another line (D=1) or plane (D=2) or hyperplane (D>2)
  for p=1:nPoints-1
    x(p,:)=x(p+1,:);
    f(p)=f(p+1);
  end
  x(nPoints,:)=xn;
  f(nPoints)=fn;

end % end of the run
% ------------------------------- Results
if ~isempty(quantis)  % Some problems may be (partly) discrete
  if ~qAlways
    xBest=quantify(xBest,quantis);
    fBest=fit(xBest,func);
    tQuantis=tQuantis+1;
  else
    fBest=fBestQ;
  end
end
tEnd=cputime;
fprintf('\n function %i, dimension %i, budget %i',func,D,tMax);
if adaptLBcoeff~=0
  switch adaptOption
    case 1
      fprintf('\n Adaptation 1')
      fprintf('\n Totally adaptive lower bound, coefficient %f',adaptLBcoeff);
```

```
    case 2
      fprintf('\n Adaptation 2');
      fprintf('\n Mix user defined lower bound %e + adaptive,  coefficient %f',...
        lowerBound,adaptLBcoeff);
      fprintf('\n (no matter the sign of fBest)')
    case 3
      fprintf('\n Adaptation 3');
      fprintf('\n Mix user defined lower bound %e + adaptive,  coefficient %f',...
        lowerBound,adaptLBcoeff);
      fprintf('\n (depending on the sign of fBest)');
  end
else
  fprintf('\n lowerBound %f',lowerBound);
end
fprintf('\n Evaluations %i', tMax+nPoints+tQuantis);
if  tQuantis>0
  fprintf(' (including %i for discretisation)',tQuantis)
end
fprintf('\n noWayCount %i',noWayCount);
fprintf('\n outsideCount %i',outsideCount);
fprintf('\n fBest %e',fBest)
fprintf('\n on \n'); disp(xBest)
fprintf('\n CPU time %e \n',tEnd-tStart)
% ================================== Plots
if plotRun figure;  end
if plotLandscape && D<=2
  fPlot(func,D,xMin,xMax,lowerBound);
  hold on
end
if plotLine
  switch D
    case 1
      plot(line(:,1), line(:,2),'r-','LineWidth',2);
      %axis([xMin,xMax,lowerBound, max(line(:,2))]);
      hold on
    case 2
      plot3(line(:,1), line(:,2),line(:,3),'r-','LineWidth',2);
      hold on
  end
end
if plotPoints
  switch D
    case 1
      plot(line(:,1), line(:,2),'r.','MarkerSize',12)
      %axis([xMin,xMax,lowerBound, max(line(:,2))]);
      hold on
```

```
    case 2
      plot3(line(:,1), line(:,2),line(:,3),'r.','MarkerSize',12)
      hold on
  end
end
if plotDash && D==1
  [tMax,~]=size(line);
  for t=1:tMax-2
    X=[line(t,1),line(t+2,1)];
    Y=[line(t,2),lowerBound];
    plot(X,Y,'r--','LineWidth',1);
    hold on
  end

  for t=1:tMax % Vertical lines
    X=[line(t,1),line(t,1)];
    Y=[lowerBound,line(t,2)];
    plot(X,Y,'r--','LineWidth',1);
    hold on
  end
  %axis([xMin,xMax,lowerBound, max(line(:,2))]);

end
end
```

```
function C=corners(xMin,xMax)
D=numel(xMin);
C=combinRepet([0,1],D);
[nPoints,~]=size(C);
for n=1:nPoints
  for d=1:D
    if C(n,d)==0
      C(n,d)=xMin(d);
    else
      C(n,d)=xMax(d);
    end
  end
end
end
```

```
function [xn,fn]=new(x,f,xMin,xMax,func,lowerBound,t)
% Construct the new point and evaluate it
global noWayCount outsideCount
```

```
% At the very beginning the intersection would necessarily be
% either "noWay" or "outside".
% We don't count it and don't check it.
if t==1
  xn=out2in(f,x,lowerBound);
  fn=fit(xn,func);
 return
end
% -------------------
[~,D]=size(x);
% Define (hyper)planes (or lines if D=1)
for n=1:D
  P=[];
  F=[];
  for d=1:D+1
    rank=n+d-1;
    P=[P;x(rank,:)];
    F=[F;f(rank)];
  end

  Plane(n,1:D+1)=plane(P,F);
end
P=[];
for n=1:D
  P=[P;Plane(n,:)];
end
[xn,noWay]=intersect(P,lowerBound);
if  noWay   % Weighted combination
  noWayCount=noWayCount+1; %Just for information
  xn=out2in(f,x,lowerBound);

else %Intersection but maybe outside the search space
  outside=false;
  for d=1:D
    if xn(d)<xMin(d) || xn(d)>xMax(d)
      outside=true;
      outsideCount=outsideCount+1;
      break;
    end
  end
  % Note: For simplicity we apply the same modification for
  %       "noWay" and "outside". You may try something else.
  if outside
   xn=out2in(f,x,lowerBound);
  end
end
```

```
 fn=fit(xn,func);
end
```

```
function xq=quantify(x,quantis)
D=numel(x);
for d=1:D
  q=quantis(d);
  if q>0
    xq(d)=q*nearest(x(d)/q);
  else
    xq(d)=x(d);
  end
end
end
```

```
function [xn,noWay]=intersect(P,lowerBound)
[nbPlans,n]=size(P);
D=n-1;
P3=[];
for np=1:nbPlans
  P3=[P3;lowerBound-P(np,n)];
end
M=[];
for np=1:nbPlans
  M=[M;P(np,1:D)];
end
%fprintf('\n M, P3 \n'); disp([M,P3])
xn=linsolve(M,P3);
% No solution
noWay=false;
check=find(xn==Inf);
%fprintf('\n check1 '); disp(check)
if ~isempty(check)
  noWay=true;
else
  check=find(xn==-Inf);
  if ~isempty(check)
    noWay=true;
  else
    for d=1:D
      if isnan(xn(d))
        noWay=true; break;
      end
```

```
    end
  end
end
```

```
function C = combinRepet(list, K)
% Combinations with repetition
% We assume K >0
nL = length(list) ;
  if K == 1  % Just the column vector
        C = list(:) ;
    else
        [L{K:-1:1}] = ndgrid(1:nL) ;
        rL = reshape(cat(K+1, L{:}), [], K) ;
        C = list(rL) ;
  end
end
```

```
function xn=out2in(f,x,lowerBound)
[~,D]=size(x);
if min(f)<0
    deltaf=f-lowerBound;
else
    deltaf=f;
end
w=sum(deltaf)-deltaf;
sw=sum(w);
for d=1:D
  xn(d)=sum(w'.*x(:,d))/sw;
end
end
```

```
function A=plane(P,F)
[n,~]=size(P);
M=[P,ones(n,1)]; % Add a column of 1
A=linsolve(M,F)';
end
```

```
function fPlot(func,D,xMin,xMax,lowerBound)
a=0.01; % For highly variable functions (like -2) use 0.001
step=a*max(xMax-xMin);
```

```
switch D
  case 1 % D=1
    X=xMin:step:xMax;
    lX=length(X);

    for i=1:lX
      Y(i)=fit(X(i),func);
    end
    plot(X,Y)
    axis([xMin,xMax,lowerBound, max(Y)]);
    %axis([xMin,xMax]);

  case 2 % D=2
    X=xMin(1):step:xMax(1);
    Y=xMin(2):step:xMax(2);

    lX=length(X);
    lY=length(Y);

    [Xx,Yy]=meshgrid(X,Y);

    for lx=1:lX
      x=X(lx);
      for ly=1:lY
        y=Y(ly);
        Z(lx,ly)=fit([x,y],func);
      end
    end
    %surf(Xx,Yy,Z');
    mesh(Xx,Yy,Z');
    xlabel("x");
    ylabel("y");
    alpha 0.01 % For transparency
end
end
```

References

Clerc, Maurice (Mar. 2024a). "Iterative optimization -Complexity and Efficiency are not antinomic." DOI: 10.13140/RG.2.2.26318.43847. URL: https://hal.science/hal-04487869 (visited on 03/11/2024).

— (2024b). *PSO programs*. URL: http://clerc.maurice.free.fr/pso/ (visited on 03/27/2024).

Das, Swagatam and P. N. Suganthan (2011). *Problem Definitions and Evaluation Criteria for CEC 2011 Competition on Testing Evolutionary Algorithms on Real World Optimization Problems*. Tech. rep.

Yang, Xin-She et al. (Jan. 2013). "True global optimality of the pressure vessel design problem: a benchmark for bio-inspired optimisation algorithms." In: *International Journal of Bio-Inspired Computation* 5.6, pp. 329–335. ISSN: 1758-0366. DOI: 10.1504/IJBIC.2013.058910. URL: http://www.inderscienceonline.com/doi/abs/10.1504/IJBIC.2013.058910 (visited on 11/05/2017).