CGO

Contents

Articles

References

CGO

Welcome to the TOMLAB /CGO User's Guide. TOMLAB /CGO includes the solvers, *rbfSolve*, *ego* and *arbfMIP*. The solvers are specifically designed to solve costly (expensive) global optimization problems with up to roughly 30 decision variables. The costly component is only the objective function, i.e. if the constraints are costly as well they need to be integrated in the objective.

The overall solution approach followed by TOMLAB /CGO is based on the seamless combination of the global and local search strategies. The package requires the presence of a global solver and a local solver.

Contents of this manual

- CGO Using the Matlab Interface provides an overview of the solver interface.
- CGO Setting Options describes how to set CGO solver options from Matlab.
- CGO Test Examples provides information regarding (non-costly) TOMLAB /CGO test examples.
- CGO Solver Reference gives detailed information about the interface routines *rbfSolve*, *ego* and *arbfMIP*.

More information

Please visit the following links for more information and see the references at the end of this manual.

- http://tomopt.com/tomlab/products/cgo/ [1]
- http://tomopt.com/tomlab/products/cgo/solvers/rbfSolve.php [2]
- http://tomopt.com/tomlab/products/cgo/solvers/ego.php [3]
- http://tomopt.com/tomlab/products/cgo/solvers/arbfMIP.php [4]

Prerequisites

In this concise manual we assume that the user is familiar with global optimization and nonlinear programming, setting up problems in TOMLAB (in particular global constrained nonlinear (**glc**) problems) and with the Matlab language in general.

Using the Matlab Interface

- Using the Matlab Interface

Setting CGO Options

- CGO Setting Options

TOMLAB /CGO Test Examples

- CGO Test Examples

TOMLAB /CGO Solver Reference

- CGO Solver Reference

rbfSolve description

- rbfSolve description

References

[1] http://tomopt.com/tomlab/products/cgo/
[2] http://tomopt.com/tomlab/products/cgo/solvers/rbfSolve.php
[3] http://tomopt.com/tomlab/products/cgo/solvers/ego.php
[4] http://tomopt.com/tomlab/products/cgo/solvers/arbfMIP.php

CGO Using the Matlab Interface

This page is part of the CGO Manual. See CGO Manual.

The CGO solver package is accessed via the *tomRun* driver routine, which calls the *rbfSolve*, *ego* or *arbfMIP* routines.

Table: The Costly global solver routines.

Function	Description
rbfSolve	Costly global solver routine called by the TOMLAB driver routine *tomRun*. This routine will also call local and global subsolvers.
ego	Costly global solver routine called by the TOMLAB driver routine *tomRun*. This routine will also call local and global subsolvers.
arbfMIP	Costly global solver routine called by the TOMLAB driver routine *tomRun*. This routine will also call local and global subsolvers.

CGO Setting Options

This page is part of the CGO Manual. See CGO Manual.

All control parameters can be set directly from Matlab.

The parameters can be set as subfields in the *Prob.CGO*, *Prob.optParam* and *Prob.GO* structures. The following example shows how to set a limit on the maximum number of iterations when using a global subsolver to solve some sub problem and the global search idea (surface search strategy) used by *rbfSolve*. The major thing is most often to set the limit *MaxFunc*, defining how many costly function evaluations the CGO solver is allowed to use.

```
Prob = glcAssign(...)            %   Setup problem, see help glcAssign for  more information


Prob.GO.MaxIter  = 50;             %   Setting the maximum number iterations.
Prob.CGO.idea = 1;              %   Idea set to first  option.
Prob.optParam.MaxFunc = 90;        %   Maximal number of  costly function evaluations
```

A complete description of the available CGO parameters can be found in CGO Solver Reference.

CGO Test Examples

This page is part of the CGO Manual. See CGO Manual.

There are several test examples included in the general TOMLAB distribution. The examples are located in the *testprob* folder in TOMLAB. *lgo1_prob* contains one dimensional test problems while *lgo2_prob* includes two- and higher-dimensional. Several problems are also available in *glb_prob*, *glc_prob*, *glcIP_prob* and *minlp_prob*.

To test the solution of these problem sets with CGO, the following type of code can be used:

```
Prob = probInit('lgo1_prob', 1);


Result  = tomRun('rbfSolve', Prob,  1);
```

CGO rbfSolve

This page is part of the CGO Manual. See CGO Manual.

Purpose

Solve general constrained mixed-integer global black-box optimization problems with costly objective functions.

The optimization problem is of the following form

$$\begin{array}{lrcl} \min\limits_{x} & f(x) & & \\ s/t & x_L & \leq \quad x \quad \leq & x_U \\ & b_L & \leq \quad Ax \quad \leq & b_U \\ & c_L & \leq \quad c(x) \quad \leq & c_U \\ & x_{j\in\mathbb{N}} & \forall j\in\mathbb{I}, & \end{array}$$

where $f(x) \in \mathbb{R}$; $x_L,\ x,\ x_U \in \mathbb{R}^d$; the $m_1$linear constraints are defined by $A \in \mathbb{R}^{m_1 \times d}$, $b_L,\ b_U \in \mathbb{R}^{m_1}$; and the $m_2$nonlinear constraints are defined by $c_L,\ c(x),\ c_U \in \mathbb{R}^{m_2}$. The variables $x_I$are restricted to be integers, where $\mathbb{I}$is an index subset of $\{1,\ldots,d\}$,possibly empty. It is assumed that the function $f(x)$is continuous with respect to all variables, even if there is a demand that some variables only take integer values. Otherwise it would not make sense to do the surrogate modeling of $f(x)$used by all CGO solvers.

$f(x)$ is assumed to be a costly function while $c(x)$ is assumed to be cheaply computed. Any costly constraints can be treated by adding penalty terms to the objective function in the following way:

$$\min_{x} \quad p(x) = f(x) + \sum_{j} w_j \max\left(0, c^j(x) - c_U^j, c_L^j - c^j(x)\right),$$

where weighting parameters w_j have been added. The user then returns $p(x)$ instead of $f(x)$ to the CGO solver.

Calling Syntax

```
Result = rbfSolve(Prob,varargin)
Result = tomRun('rbfSolve', Prob);
```

Description of Inputs

Problem description structure. The following fields are used:

Field*NameFUNCS.fFUNCS.cx_Lx_Ub_Ub_LAc_Lc_UWarmStartMaxCPUuserPriLevOptPriLevSubf_LowoptParamMaxFuncIterPr*
bTol
cTolMaxIterCGOPercentnSampleX,F,CXXFCXRandStateAddMPnTrialCLHMethodSCALEREPLACELOCALSMOOTHglobalSolver
Name of the problem. Used for security when doing warm starts. Name of function to compute the objective function. Name of function to compute the nonlinear constraint vector. Lower bounds on the variables. Must be finite. Upper bounds on the variables. Must be finite. Upper bounds for the linear constraints. Lower bounds for the linear constraints. Linear constraint matrix. Lower bounds for the nonlinear constraints. Upper bounds for the nonlinear constraints. Set true (non-zero) to load data from previous run from *cgoSave.mat* and resume optimization from where the last run ended. If *Prob.CGO.WarmStartInfo* has been defined through a call to *WarmDefGLOBAL*, this field is used instead of the *cgoSave.mat* file. All CGO solvers uses the same mat-file and structure field and can read the output of one another.Maximal CPU Time (in seconds) to be used. User field used to send information to low-level functions. Print Level. 0 = silent. 1 = Summary 2 = Printing each iteration. 3 = Info about local / global solution. 4 = Progress in x. Print Level in subproblem solvers, see help in *snSolve* and *gnSolve*.Lower bound on the optimal function value. If defined, used to restrict the target values into interval \[f Low,min(surface)\]. Structure

with optimization parameters. The following fields are used: Maximal number of costly function evaluations, default 300 for *rbfSolve* and *arbfMIP*, and default 200 for *ego*. *MaxFunc* must be <= 5000. If WarmStart = 1 and MaxFunc <= nFunc (Number of f(x) used) then set MaxFunc := MaxFunc + nFunc.Print one information line each iteration, and the new x tried. Default IterPrint = 1. fMinI means the best f(x) is infeasible. fMinF means the best f(x) is feasible (also integer feasible). Goal for function value, not used if *inf* or empty.Relative accuracy for function value, *fTol* == *eps_f*. Stop if |*f* - *f Goal*| <= |*fGoal*| * *fTol*, if *fGoal* ≠ 0. Stop if |*f* - *fGoal*| <= *fTol*, if *fGoal* = 0. See the output field maxTri. Linear constraint tolerance. Nonlinear constraint tolerance. Maximal number of iterations used in the local optimization on the re- sponse surface in each step. Default 1000, except for pure IP problems, then max(GO.MaxFunc, MaxIter);. Structure (*Prob.CGO*) with parameters concerning global optimization options.

Type of strategy to get the initial sampled values:

Percent	Experimental Design	ExD
	Corner strategies	
900	All Corners	1
997	$x_L + x_U$ + adjacent corners	2
998	x_U + adjacent corners	3
999	x_L + adjacent corners	4
	Deterministic Strategies	
0	User given initial points	5
94	DIRECT solver *glbFast*	6
95	DIRECT solver *glcFast*	6
96	DIRECT solver *glbSolve*	6
97	DIRECT solver *glcSolve*	6
98	DIRECT solver *glbDirect*	6
99	DIRECT solver *glcDirect*	6
	Latin Based Sampling	
1	Maximin LHD 1-norm	7
2	Maximin LHD 2-norm	8
3	Maximin LHD Inf-norm	9
4	Minimal Audze-Eglais	10
5	Minimax LHD (only 2 dim)	11
6	Latin Hypercube	12
7	Orthogonal Samling	13
	Random Strategies (pp in %)	
1pp	Circle surrounding	14
2pp	Ellipsoid surrounding	15
3pp	Rectangle surrounding	16

Negative values of Percent result in constrained versions of the experimental design methods 7-16. It means that all points sampled are feasible with respect to all given constraints.

For ExD 5,6-12,14-16 user defined points are used.

Number of sample points to be used in initial experimental design. *nSample* is used differently dependent on the value of Percent:

	(n)Sample:			
ExD	**< 0**	**= 0**	**> 0**	**[]**
1	2^d			
6	|n| iterations			
7-11	d+1	d+1	max(d + 1,n)	(d + 1)(d + 2) / 2
12	LATIN(k)			
13	|n|			
14-16	d + 1			

where LATIN = [21 21 33 41 51 65 65] and k = |*nSample*|. Otherwise nSample as input does not matter.

Description of the experimental designs:

ExD 1, All Corners. Initial points is the corner points of the box given by Prob.x_L and Prob.x_U. Generates 2^d points, which results in too many points when the dimension is high.

ExD 2, Lower and Upper Corner point + adjacent points. Initial points are 2 * d + 2 corners: the lower left corner x_L and its d adjacent corners x_L + (x_U(*i*) - x_L (i)) * e_i, i = *1*, ..., d and the upper right corner x_U and its d adjacent corners x_U - (x_U (i) - x_L (i)) * e_i, i = *1*, ..., d

ExD 3. Initial points are the upper right corner x_U and its d adjacent corners x_U - (x_U (*i*) - x_L (*i*)) * e_i, *i* = 1, ..., *d*

ExD 4. Initial points are the lower left corner x_L and its d adjacent corners x_L + (x_U (*i*) - x_L (*i*)) * e_i, *i* = 1, ..., *d*

ExD 5. User given initial points, given as a matrix in CGO.X. Each column is one sampled point. If d >= length(Prob.x L), then size(X,1) = d, size(X,2) = d + 1. CGO.F should be defined as empty, or contain a vector of corresponding $f(x)$ values. Any CGO.F value set as NaN will be computed by solver routine.

ExD 6. Use determinstic global optimization methods to find the initial design. Current methods available (all DIRECT methods), dependent on the value of Percent:

99 = glcDirect, 98 = glbDirect, 97 = glcSolve, 96 = glbSolve, 95 = glcFast, 94 = glbFast.

ExD 7-11. Optimal Latin Hypercube Designs (LHD) with respect to different norms. The following norms and designs are available, dependent on the value of Percent:

1 = Maximin 1-Norm, 2 = Maximin 2-Norm, 3 = Maximin Inf-Norm, 4 = Audze-Eglais Norm, 5 = Minimax 2-Norm.

All designs taken from: http://www.spacefillingdesigns.nl/ [1]

Constrained versions will try bigger and bigger designs up to M = max(10 * *d, nTrial*) different designs, stopping when it has found nSample feasible points.

ExD 12. Latin hypercube space-filling design. For nSample < 0, k = |*nSample*| should in principle be the problem dimension. The number of points sampled is:

k : 2 3 4 5 6 > 6

Points : 21 33 41 51 65 65

The call made is: X = daceInit(abs(nSample),Prob.x_L,Prob.x_U);

Set nSample = [] to get (d+1)*(d+2)/2 sampled points:

d : 1 2 3 4 5 6 7 8 9 10

Points : 3 6 10 15 21 28 36 45 55 66

This is a more efficient number of points to use.

If CGO.X is nonempty, these points are verified as in ExD 5, and treated as already sampled points. Then nSample additional points are sampled, restricted to be close to the given points.

Constrained version of Latin hypercube only keep points that fulfill the linear and nonlinear constraints. The algorithm will try up to $M = max(10 * d, nTrial)$ points, stopping when it has found nSample feasible points ($d + 1$ points if $nSample < 0$).

ExD 13. Orthogonal Sampling, LH with subspace density demands.

ExD 14-16. Random strategies, the |*Percent*| value gives the percentage size of an ellipsoid, circle or rectangle around the so far sampled points that new points are not allowed in. Range 1%-50%. Recommended values 10% - 20%. If CGO.X is nonempty, these points are verified as in ExD 5, and treated as already sampled points. Then nSample additional points are sampled, restricted to be close to the given points.

The fields X,F,CX are used to define user given points. ExD = 5 (Percent = 0) needs this information. If ExD == 6-12,14-16 these points are included into the design. A matrix of initial x values. One column for every x value. If ExD == 5, size(X,2) >= dim(x)+1 needed.A vector of initial $f(x)$ values. If any element is set to NaN it will be computed.Optionally a matrix of nonlinear constraint c(x) values. If nonempty, then size(CX,2) == size(X,2). If any element is set as NaN, the vector c(x) = CX(:,i) will be recomputed. If >= 0, *rand*(*'state', RandState*) is set to initialize the pseudo-random generator. If < 0, *rand*(*'state',* 100 * *clock*) is set to give a new set of random values each run. If isnan(RandState), the random state is not initialized. RandState will influence if a stochastic initial experimental design is applied, see input Percent and nSample. RandState will also influence if using the *multiMin* solver, but the random state seed is not reset in *multiMin*. The state of the random generator is saved in the warm start output rngState, and the random generator is reinitialized with this state if warm start is used. Default RandState = 0.If = 1, add the midpoint as extra point in the corner strategies. Default 1 for any corner strategy, i.e. Percent is 900, 997, 998 or 999. For experimental design CLH, the method generates $M = max(10 * d, nTrial)$ trial points, and evaluate them until *nSample* feasible points are found. In the random designs, *nTrial* is the maximum number of trial points randomly generated for each new point to sample.

Different search strategies for finding feasible LH points. First of all, the least infeasible point is added. Then the linear feasible points are considered. If more points are needed still, the nonlinear infeasible points are added.

1 - Take the sampled infeasible points in order.

2 - Take a random sample of the infeasible points.

3 - Use points with lowest constraint error (cErr).

0 - Original search space (default if any integer values).

1 - Transform search space to unit cube (default if no integers).

0 - No replacement, default for constrained problems.

1 - Large function values are replaced by the median.

> 1 - Large values Z are replaced by new values. The replacement is defined as $Z := FMAX + log10(Z - FMAX + 1)$, where $FMAX = 10^{REPLACE}$, if $min(F) < 0$ and $FMAX = 10^{(ceil(log10(min(F))) + REPLACE)}$, if $min(F) = 0$. A new replacement is computed in every iteration, because $min(F)$ may change. Default REPLACE = 5, if no linear or nonlinear constraints.

0 - No local searches after global search. If RBF surface is inaccurate, might be an advantage.

1 - Local search from best points after global search. If equal best function values, up to 20 local searches are done.

1 - The problem is smooth enough for local search using numerical gradient estimation methods (default).

0 - The problem is nonsmooth or noisy, and local search methods using numer- ical gradient estimation are likely to produce garbage search directions.

Global optimization solver used for subproblem optimization. Default *glcCluster* (SMOOTH=1) or *glcDirect* (SMOOTH=0). If the global- Solver is *glcCluster*, the fields *Prob.GO.maxFunc1*, *Prob.GO.maxFunc2*, *Prob.GO.maxFunc3*, *Prob.GO.localSolver*, *Prob.GO.DIRECT* and other fields set in *Prob.GO* are used. See the help for these parameters in *glcCluster*.

Local optimization solver used for subproblem optimization. If not defined, the TOMLAB default constrained NLP solver is used.

- Special RBF algorithm parameters in Prob.CGO -

Type of radial basis function: 1 - thin plate spline; 2 - Cubic Spline (default); 3 - Multiquadric; 4 - Inverse multiquadric; 5 - Gaussian; 6 - Linear.

Type of search strategy on the response surface.

idea = 1 - cycle of N+1 points in target value *fnStar*.

if *fStarRule* =3, then N=1 default, otherwise N=4 default.

By default *idea* =1, *fStarRule* =1, i.e. *N* =4. To change *N*, see below.

idea = 2 - cycle of 4 points (N+1, N=3 always) in *alpha*. *alpha* is a bound on an algorithmic constraint that implicitly sets a target value *fStar*.

Cycle length in idea 1 (default N=1 for fStarRule 3, otherwise default N=4) or idea 2 (always N=3).

If =1, add search step with target value *-8* first in cycle. Default 0. Always

=1 for the case *idea* =1, *fStarRule* =3.

Global-Local search strategy in idea 1, where N is the cycle length. Define *minsn* as the global minimum on the RBF surface. The following strategies for setting the target value *fStar* is defined: 1: $fStar = min_{sn} - ((N - (n - nInit))/N)^2 * \Delta n$ (Default), 2: $fStar = min_{sn} - (N - (n - nInit))/N * \Delta n$.

Strategy 1 and 2 depends on Δ_n estimate (see DeltaRule). If *infStep* =1, add $-\infty$-step first in cycle. 3: fStar = $-\infty$ -step, min_{sn}-*k* *0.1*$|min_{sn}|k = N, ..., 0$.

These strategies had the following names in Gutmanns thesis: III, II, I.

1 = Skip large f(x) when computing f(x) interval ?. 0 = Use all points. Default 1. Add up to AddSurfMin interior local minima on RBF surface as search points, based on estimated Lipschitz constants. AddSurfMin=0 implies no additional minimum added (Default). This option is only possible if *globalSolver* = *multiMin*. Test for additional minimum is done in the local step (modN == N) If these additional local minima are used, in the printout modN = -2, -3, -4, ... are the iteration steps with these search points.

Which minimum, if several minima found, to select in the target value problem:

=0 Use global minimum.

=1 Use best interior local minima, if none use global minimum.

=2 Use best interior local minima, if none use RBF interior minimum.

=3 Use best minimum with lowest number of coefficients on bounds.

Default is *TargetMin* = 3.

Relative tolerance used to test if the minimum of the RBF surface, min_{sn} , is sufficiently lower than the best point (*fM in*) found (default is 10^{-7}).Max number of cycles without progress before stopping, default 10.

Structure *Prob.GO* (Default values are set for all fields).

The following fields are used:

Maximal number of function evaluations in each global search. Maximal number of iterations in each global search. DIRECT solver used in glcCluster, either glcSolve or glcDirect(default). glcCluster parameter, maximum number of function evaluations in the first call. Only used if globalSolver is glcCluster, see help globalSolver. glcCluster

parameter, maximum number of function evaluations in the second call. Only used if globalSolver is glcCluster, see help globalSolver. glcCluster parameter, maximum sum of function evaluations in repeated first calls to DIRECT routine when trying to get feasible. Only used if globalSolver is glcCluster, see help *globalSolver*.The local solver used by glcCluster. If not defined, then *Prob.CGO.localSolver* is used

Structure in Prob, Prob.MIP.

Defines integer optimization parameters. Fields used:

If empty, all variables are assumed non-integer.

If islogical(IntVars) (=all elements are 0/1), then 1 = integer variable, 0 = continuous variable. If any element > 1, IntVars is the indices for integer variables.

Other parameters directly sent to low level routines.

Description of Outputs

Structure with result from optimization. The following fields are changed:

Field	Description
x_k	Matrix with the best points as columns.
f_k	The best function value found so far.
Iter	Number of iterations.
FuncEv	Number of function evaluations.
ExitText	Text string with information about the run.
ExitFlag	Always 0.
CGO	Subfield *WarmStartInfo* saves warm start information, the same information as in cgoSave.mat, see below.
Inform	Information parameter. 0 = Normal termination. 1 = Function value f(x) is less than fGoal. 2 = Error in function value $f(x)$, $\|f - fGoal\| <= fTol$, $fGoal = 0$. 3 = Relative Error in function value $f(x)$ is less than fTol, i.e. |f - fGoal| |fGoal| <= fTol. 4 = No new point sampled for MaxCycle iteration steps. 5 = All sample points same as the best point for MaxCycle last iterations. 6 = All sample points same as previous point for MaxCycle last iterations. 7 = All feasible integers tried. 8 = No progress for $MaxCycle * (N + 1) + 1$ function evaluations (> *MaxCycle* cycles, input CGO.MaxCycle). 9 = Max CPU Time reached.
cgoSave.mat	To make a warm start possible, all CGO solvers saves information in the file cgoSave.mat. The file is created independent of the solver, which enables the user to call any CGO solver using the warm start information. cgoSave.mat is a MATLAB mat-file saved to the current directory. If the parameter SAVE is 1, the CGO solver saves the mat file every iteration, which enables the user to break the run and restart using warm start from the current state. SAVE = 1 is currently always set by the CGO solvers. If the cgoSave.mat file fails to open for writing, the information is also available in the output field Result.CGO.WarmStartInfo, if the run was concluded without interruption. Through a call to WarmDefGLOBAL, the Prob structure can be setup for warm start. In this case, the CGO solver will not load the data from cgoSave.mat. The file contains the following variables:
Name	Problem name. Checked against the *Prob.Name* field if doing a warmstart.
O	Matrix with sampled points (in original space).
X	Matrix with sampled points (in unit space if SCALE==1)
F	Vector with function values (penalty added for costly Cc(x))

F_m	Vector with function values (replaced).
F00	Vector of pure function values, before penalties. *Cc* MMatrix with costly constraint values, *C* $c(x)$. *nInit* Number of initial points.
Fpen	Vector with function values + additional penalty if infeasible using the linear constraints and noncostly nonlinear $c(x)$.
fMinIdx	Index of the best point found.
rngState	Current state of the random number generator used.

Description

rbfSolve implements the Radial Basis Function (RBF) algorithm based on the work by Gutmann. The RBF method is enhanced to handle linear equality and inequality constraints, and nonlinear equality and inequality constraints, as well as mixed-integer problems.

A response surface based on radial basis functions is fitted to a collection of sampled points. The algorithm then balances between minimizing the fitted function and adding new points to the set.

M-files Used

daceInit.m, *iniSolve.m*, *endSolve.m*, *conAssign.m*, *glcAssign.m*, *snSolve.m*, *gnSolve.m*, *expDesign.m*.

MEX-files Used

tomsol

See Also

ego.m

Warnings

Observe that when cancelling with CTRL+C during a run, some memory allocated by rbfSolve will not be deallocated. To deallocate, do:

```
>> clear cgolib
```

References

[1] http://www.spacefillingdesigns.nl/

CGO ego

This page is part of the CGO Manual. See CGO Manual.

Purpose

Solve general constrained mixed-integer global black-box optimization problems with costly objective functions.

The optimization problem is of the following form

$$\begin{array}{lrcccl} \min\limits_{x} & f(x) & & & & \\ s/t & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \\ & x_j \in \mathbb{N} \;\; \forall j \in \mathbb{I} & & & & \end{array},$$

where $f(x) \in \mathbb{R}$; $x_L,\ x,\ x_U \in \mathbb{R}^d$; the $m_1$linear constraints are defined by $A \in \mathbb{R}^{m_1 \times d}$, $b_L,\ b_U \in \mathbb{R}^{m_1}$; and the $m_2$nonlinear constraints are defined by $c_L,\ c(x),\ c_U \in \mathbb{R}^{m_2}$. The variables $x_I$are restricted to be integers, where $\mathbb{I}$is an index subset of $\{1, \ldots, d\}$,possibly empty. It is assumed that the function $f(x)$is continuous with respect to all variables, even if there is a demand that some variables only take integer values. Otherwise it would not make sense to do the surrogate modeling of $f(x)$used by all CGO solvers.

$f(x)$ is assumed to be a costly function while $c(x)$ is assumed to be cheaply computed. Any costly constraints can be treated by adding penalty terms to the objective function in the following way:

$$\min_{x} \quad p(x) = f(x) + \sum_{j} w_j \max\left(0,\ c^j(x) - c_U^j,\quad c_L^j - c^j(x)\right),$$

where weighting parameters w_j have been added. The user then returns $p(x)$ instead of $f(x)$ to the CGO solver.

Calling Syntax

```
Result=ego(Prob,varargin)
Result = tomRun('ego', Prob);
```

Description of Inputs

Problem description structure. The following fields are used:

Field*NameFUNCS.fFUNCS.cx_Lx_Ub_Ub_LAc_Lc_UWarmStartMaxCPUuserPriLevOptPriLevSuboptParamMaxFuncIterPrintfGo*
Name of the problem. Used for security when doing warm starts. Name of function to compute the objective function. Name of function to compute the nonlinear constraint vector. Lower bounds on the variables. Must be finite. Upper bounds on the variables. Must be finite. Upper bounds for the linear constraints. Lower bounds for the linear constraints. Linear constraint matrix. Lower bounds for the nonlinear constraints. Upper bounds for the nonlinear constraints. Set true (non-zero) to load data from previous run from *cgoSave.mat* and re- sume optimization from where the last run ended. If *Prob.CGO.WarmStartInfo* has been defined through a call to *WarmDefGLOBAL*, this field is used instead of the *cgoSave.mat* file. All CGO solvers uses the same mat-file and structure field and can read the output of one another.Maximal CPU Time (in seconds) to be used. User field used to send information to low-level functions. Print level. 0 = silent. 1 = Summary, 2 = Printing each iteration, 3 = Info about local / global solution, 4 = Progress in x. Print Level in subproblem solvers. Structure with optimization parameters. The following fields are used: Maximal number of costly function evaluations, default 300 for *rbfSolve* and *arbfMIP*, and default 200 for *ego*. *MaxFunc* must be <= 5000. If WarmStart = 1 and MaxFunc <= nFunc (Number of f(x) used) then set MaxFunc := MaxFunc + nFunc.Print one information line each iteration, and the new

x tried. Default IterPrint = 1. fMinI means the best f(x) is infeasible. fMinF means the best f(x) is feasible (also integer feasible). Goal for function value, not used if *inf* or empty.Relative accuracy for function value, *fTol == eps_f* . Stop if $|f - f\ Goal| = |f\ Goal| * fTol$, if $f\ Goal = 0$. Stop if $|f - f\ Goal| = fTol$, if $f\ Goal = 0$. See the output field maxTri. Linear constraint tolerance. Nonlinear constraint tolerance. Maximal number of iterations used in the local optimization on the re- sponse surface in each step. Default 1000, except for pure IP problems, then max(GO.MaxFunc, MaxIter);.

Structure (*Prob.CGO*) with parameters concerning global optimization options.

The following general fields in Prob.CGO are used:

Type of strategy to get the initial sampled values:

Percent	Experimental Design	ExD
	Corner strategies	
900	All Corners	1
997	$x_L + x_U$ + adjacent corners	2
998	x_U + adjacent corners	3
999	x_L + adjacent corners	4
	Deterministic Strategies	
0	User given initial points	5
94	DIRECT solver *glbFast*	6
95	DIRECT solver *glcFast*	6
96	DIRECT solver *glbSolve*	6
97	DIRECT solver *glcSolve*	6
98	DIRECT solver *glbDirect*	6
99	DIRECT solver *glcDirect*	6
	Latin Based Sampling	
1	Maximin LHD 1-norm	7
2	Maximin LHD 2-norm	8
3	Maximin LHD Inf-norm	9
4	Minimal Audze-Eglais	10
5	Minimax LHD (only 2 dim)	11
6	Latin Hypercube	12
7	Orthogonal Samling	13
	Random Strategies (pp in %)	
1pp	Circle surrounding	14
2pp	Ellipsoid surrounding	15
3pp	Rectangle surrounding	16

Negative values of Percent result in constrained versions of the experimental design methods 7-16. It means that all points sampled are feasible with respect to all given constraints.

For ExD 5,6-12,14-16 user defined points are used.

Number of sample points to be used in initial experimental design. *nSample* is used differently dependent on the value of Percent:

	(n)Sample:	ExD	< 0	= 0	> 0	[]
1	2^d					
6	|n|					
7-11	$d + 1$	$d + 1$	max $(d + 1, n)$	$(d + 1)(d + 2)/2$		
12	LATIN(k)					
13	|n|					
14-16	$d + 1$					

where LATIN = [21 21 33 41 51 65 65] and k = |*nSample*|. Otherwise nSample as input does not matter.

Description of the experimental designs:

ExD 1, All Corners. Initial points is the corner points of the box given by Prob.x L and Prob.x U. Generates 2d points, which results in too many points when the dimension is high.

ExD 2, Lower and Upper Corner point + adjacent points. Initial points are 2 * d + 2 corners: the lower left corner x_L and its d adjacent corners $x_L + (x_U(i) - x_L(i)) * e_i$, $i = 1, ..., d$ and the upper right corner x_U and its d adjacent corners $x_U - (x_U(i) - x_L(i)) * e_i$, $i = 1, ..., d$

ExD 3. Initial points are the upper right corner x_U and its d adjacent corners $x_U - (x_U(i) - x_L(i)) * e_i$, $i = 1, ..., d$

ExD 4. Initial points are the lower left corner x_L and its d adjacent corners $x_L + (x_U(i) - x_L(i)) * e_i$, $i = 1, ..., d$

ExD 5. User given initial points, given as a matrix in CGO.X. Each column is one sampled point. If d = length(Prob.x_L), then size(X,1) = d, size(X,2) = d + 1. CGO.F should be defined as empty, or contain a vector of corresponding $f(x)$ values. Any CGO.F value set as NaN will be computed by solver routine.

ExD 6. Use determinstic global optimization methods to find the initial design. Current methods available (all DIRECT methods), dependent on the value of Percent:

99 = glcDirect, 98 = glbDirect, 97 = glcSolve, 96 = glbSolve, 95 = glcFast, 94 = glbFast.

ExD 7-11. Optimal Latin Hypercube Designs (LHD) with respect to different norms. The following norms and designs are available, dependent on the value of Percent:

1 = Maximin 1-Norm, 2 = Maximin 2-Norm, 3 = Maximin Inf-Norm, 4 = Audze-Eglais Norm, 5 = Minimax 2-Norm.

All designs taken from: http://www.spacefillingdesigns.nl/ [1]

Constrained versions will try bigger and bigger designs up to M = max(10 * *d, nT rial*) different designs, stopping when it has found nSample feasible points.

ExD 12. Latin hypercube space-filling design. For nSample < 0, k = |*nSample*| should in principle be the problem dimension. The number of points

sampled is:

k : 2 3 4 5 6 > 6

Points : 21 33 41 51 65 65

The call made is: X = daceInit(abs(nSample),Prob.x L,Prob.x U);

Set nSample = [] to get (d+1)*(d+2)/2 sampled points:

d : 1 2 3 4 5 6 7 8 9 10

Points : 3 6 10 15 21 28 36 45 55 66

This is a more efficient number of points to use.

If CGO.X is nonempty, these points are verified as in ExD 5, and treated as already sampled points. Then nSample additional points are sampled, restricted to be close to the given points.

Constrained version of Latin hypercube only keep points that fulfill the linear and nonlinear constraints. The algorithm will try up to $M = max(10 * d, nTrial)$ points, stopping when it has found nSample feasible points ($d + 1$ points if $nSample < 0$).

ExD 13. Orthogonal Sampling, LH with subspace density demands.

ExD 14-16. Random strategies, the |*Percent*| value gives the percentage size of an ellipsoid, circle or rectangle around the so far sampled points that new points are not allowed in. Range 1%-50%. Recommended values 10% - 20%.

If CGO.X is nonempty, these points are verified as in ExD 5, and treated as already sampled points. Then nSample additional points are sampled, restricted to be close to the given points.

The fields X,F,CX are used to define user given points. ExD = 5 (Percent = 0) needs this information. If ExD == 6-12,14-16 these points are included into the design. A matrix of initial x values. One column for every x value. If ExD == 5, size(X,2) >= dim(x)+1 needed.A vector of initial $f(x)$ values. If any element is set to NaN it will be computed.Optionally a matrix of nonlinear constraint c(x) values. If nonempty, then size(CX,2) == size(X,2). If any element is set as NaN, the vector c(x) = CX(:,i) will be recomputed. If = 0, *rand*(*'state', RandState*) is set to initialize the pseudo-random generator. If < 0, *rand*(*'state',* 100 * *clock*) is set to give a new set of random values each run. If isnan(RandState), the random state is not initialized. RandState will influence if a stochastic initial experimental design is applied, see input Percent and nSample. RandState will also influence if using the *multiMin* solver, but the random state seed is not reset in *multiMin*. The state of the random generator is saved in the warm start output rngState, and the random generator is reinitialized with this state if warm start is used. Default RandState = 0.If = 1, add the midpoint as extra point in the corner strategies. Default 1 for any corner strategy, i.e. Percent is 900, 997, 998 or 999. For experimental design CLH, the method generates $M = max(10 * d, nTrial)$ trial points, and evaluate them until *nSample* feasible points are found. In the random designs, *nTrial* is the maximum number of trial points randomly generated for each new point to sample.

Different search strategies for finding feasible LH points. First of all, the least infeasible point is added. Then the linear feasible points are considered. If more points are needed still, the nonlinear infeasible points are added.

1 - Take the sampled infeasible points in order.

2 - Take a random sample of the infeasible points.

3 - Use points with lowest constraint error (cErr).

0 - Original search space (default if any integer values).

1 - Transform search space to unit cube (default if no integers).

0 - No replacement, default for constrained problems.

1 - Large function values are replaced by the median.

> 1 - Large values Z are replaced by new values. The replacement is defined as $Z := FMAX + log10(Z - FMAX + 1)$, where $FMAX = 10^{REPLACE}$, if $min(F) < 0$ and $FMAX = 10^{(ceil(log10(min(F)))+REPLACE)}$, if $min(F) >= 0$. A new replacement is computed in every iteration, because $min(F)$ may change. Default REPLACE = 5, if no linear or nonlinear constraints.

0 - No local searches after global search. If RBF surface is inaccurate, might be an advantage.

1 - Local search from best points after global search. If equal best function values, up to 20 local searches are done.

1 - The problem is smooth enough for local search using numerical gradient estimation methods (default).

0 - The problem is nonsmooth or noisy, and local search methods using numer- ical gradient estimation are likely to produce garbage search directions.

Global optimization solver used for subproblem optimization. Default *glcCluster* (SMOOTH=1) or *glcDirect* (SMOOTH=0). If the global- Solver is *glcCluster*, the fields *Prob.GO.maxFunc1*, *Prob.GO.maxFunc2*, *Prob.GO.maxFunc3*, *Prob.GO.localSolver*, *Prob.GO.DIRECT* and other fields set in *Prob.GO* are used. See the help for these parameters in *glcCluster*.

Local optimization solver used for subproblem optimization. If not defined, the TOMLAB default constrained NLP solver is used.

- Special EGO algorithm parameters in Prob.CGO -

Main algorithm in the EGO solver (default EGOAlg == 1)

=1 Run expected improvement steps (modN=0,1,2,...). If no $f(x)$ improve- ment, use DACE surface minimum (modN=-1) in 1 step

=2 Run expected improvement steps (modN=0) until ExpI/-yMin- ¡ Tol- ExpI for 3 successive steps (modN=1,2,3) without $f(x)$ improvement (fRed = 0), where yMin is fMin transformed by TRANSFORM After 2 such steps (when modN=2), 1 step using the DACE surface minimum (modN=-1) is tried. If then fRed ¿0, reset to modN=0 steps.

1 - Estimate d-vector, p parameters (default), 0 - fix p=2.

Norm parameters, fixed or estimated, also see p0, pLow, pUpp (default pEst = 0).

0 = Fixed constant p-value for all components (default, p0=1.99).

1 = Estimate one p-value valid for all components.

> 1 = Estimate d|||| $_p$ *parameters, one for each component.*

Fixed p-value (pEst==0, default = 1.99) or initial p-value (pEst == 1, default 1.9) or d-vector of initial p-values (pEst > 1, default 1.9*ones(d,1))

Lower bound on p.

If pEst == 0, not used

if pEst == 1, lower bound on p-value (default 1.0)

if pEst > 1, lower bounds on p (default ones(d,1))

Upper bound on p.

If pEst == 0, not used

if pEst == 1, upper bound on p-value (default 2.0)

if pEst > 1, upper bounds on p (default 2*ones(d,1))

Function value transformation.

0 - No transformation made.

1 - Median value transformation. Use REPLACE instead.

2 - log(y) transformation made.

3 - -log(-y) transformation made.

4 - -1/y transformation made.

Default EGO is computing the best possible transformation from the initial set of data. Note! No check is made on illegal y if user gives TRANSFORM.

Transformation of expected improvement function (default 1).

= 0 No transformation made.

= 1 - log(-f) transformation made.

= 2 -1/f transformation made.

Convergence tolerance for expected improvement (default 10-6).

Sample criterion function:

0 = Expected improvment (default)

1 = Kushner's criterion (related option: KEPS)

2 = Lower confidence bounding (related option: LCBB)

3 = Generalized expected improvement (related option: GEIG)

4 = Maximum variance

5 = Watson and Barnes 2

K EP S\| * fM in .The exponent *g* in the generalized expected improvement function (default 2.0).Lower Confidence Bounding parameter b (default 2.0).

Structure *Prob.GO* (Default values are set for all fields).

The following fields are used:

Maximal number of function evaluations in each global search. Maximal number of iterations in each global search. DIRECT solver used in glcCluster, either glcSolve or glcDirect(default). glcCluster parameter, maximum number of function evaluations in the first call. Only used if globalSolver is glcCluster, see help globalSolver. glcCluster parameter, maximum number of function evaluations in the second call. Only used if globalSolver is glcCluster, see help globalSolver. glcCluster parameter, maximum sum of function evaluations in repeated first calls to DIRECT routine when trying to get feasible. Only used if globalSolver is glcCluster, see help *globalSolver*.The local solver used by glcCluster. If not defined, then *Prob.CGO.localSolver* is used

Structure in Prob, Prob.MIP.

Defines integer optimization parameters. Fields used:

If empty, all variables are assumed non-integer.

If islogical(IntVars) (=all elements are 0/1), then 1 = integer variable, 0 = continuous variable. If any element > 1, IntVars is the indices for integer variables.

Other arguments sent directly to low level functions.

Description of Outputs

Structure with result from optimization.

Output	Description
x_k	Matrix with the best points as columns.
f_k	The best function value found so far.
Iter	Number of iterations.
FuncEv	Number of function evaluations.
ExitText	Text string with information about the run.
ExitFlag	Always 0.
CGO	Subfield *WarmStartInfo* saves warm start information, the same information as in cgoSave.mat, see below.

Inform	Information parameter. 0 = Normal termination. 1 = Function value $f(x)$ is less than fGoal. 2 = Error in function value $f(x)$, $abs(f - fGoal) <= fTol$, fGoal=0. 3 = Relative Error in function value f(x) is less than fTol, i.e. $abs(f - fGoal)/abs(fGoal) <= fTol$. 4 = No new point sampled for *N* iteration steps. 5 = All sample points same as the best point for *N* last iterations. 6 = All sample points same as previous point for *N* last iterations. 7 = All feasible integers tried. 9 = Max CPU Time reached. 10 = Expected improvement low for three iterations.
Result	Structure with result from optimization.
cgoSave.mat	To make a warm start possible, all CGO solvers saves information in the file cgoSave.mat. The file is created independent of the solver, which enables the user to call any CGO solver using the warm start information. cgoSave.mat is a MATLAB mat-file saved to the current directory. If the parameter SAVE is 1, the CGO solver saves the mat file every iteration, which enables the user to break the run and restart using warm start from the current state. SAVE = 1 is currently always set by the CGO solvers. If the cgoSave.mat file fails to open for writing, the information is also available in the output field Result.CGO.WarmStartInfo, if the run was concluded without interruption. Through a call to WarmDefGLOBAL, the Prob structure can be setup for warm start. In this case, the CGO solver will not load the data from cgoSave.mat. The file contains the following variables:
Name	Problem name. Checked against the *Prob.Name* field if doing a warmstart.
O	Matrix with sampled points (in original space).
X	Matrix with sampled points (in unit space if SCALE==1)
F	Vector with function values (penalty added for costly Cc(x))
F_m	Vector with function values (replaced).
F00	Vector of pure function values, before penalties.
Cc	MMatrix with costly constraint values, $C\ c(x)$.
nInit	Number of initial points.
Fpen	Vector with function values + additional penalty if infeasible using the linear constraints and noncostly nonlinear $c(x)$.
fMinIdx	Index of the best point found.
rngState	Current state of the random number generator used.

Description

ego implements the algorithm EGO by D. R. Jones, Matthias Schonlau and William J. Welch presented in the paper "Efficient Global Optimization of Expensive Black-Box Functions".

Please note that Jones et al. has a slightly different problem formulation. The TOMLAB version of *ego* treats linear and nonlinear constraints separately.

ego samples points to which a response surface is fitted. The algorithm then balances between sampling new points and minimization on the surface.

ego and *rbfSolve* use the same format for saving warm start data. This means that it is possible to try one solver for a certain number of iterations/function evaluations and then do a warm start with the other. Example:

```
>> Prob      = probInit('glc_prob',1);            %   Set up problem structure
>> Result_ego = tomRun('ego',Prob);             %   Solve for a while with  ego
>> Prob.WarmStart = 1;                          %   Indicate a warm start
>> Result_rbf = tomRun('rbfSolve',Prob);       %   Warm start with rbfSolve
```

M-files Used

iniSolve.m, *endSolve.m*, *conAssign.m*, *glcAssign.m*

See Also

rbfSolve

Warnings

Observe that when cancelling with CTRL+C during a run, some memory allocated by ego will not be deallocated.

To deallocate, do:

```
>> clear egolib
```

CGO arbfMIP

This page is part of the CGO Manual. See CGO Manual.

Purpose

Solve general constrained mixed-integer global black-box optimization problems with costly objective functions.

The optimization problem is of the following form

$$\begin{array}{lrcccl} \min\limits_{x} & f(x) & & & & \\ s/t & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \\ & x_j \in \mathbb{N} \quad \forall j \in \mathbb{I}, & & & & \end{array}$$

where $f(x) \in \mathbb{R}$; $x_L,\ x,\ x_U \in \mathbb{R}^d$; the $m_1$linear constraints are defined by $A \in \mathbb{R}^{m_1 \times d}$, $b_L,\ b_U \in \mathbb{R}^{m_1}$; and the $m_2$nonlinear constraints are defined by $c_L,\ c(x),\ c_U \in \mathbb{R}^{m_2}$. The variables $x_I$are restricted to be integers, where $\mathbb{I}$is an index subset of $\{1, \ldots, d\}$,possibly empty. It is assumed that the function $f(x)$is continuous with respect to all variables, even if there is a demand that some variables only take integer values. Otherwise it would not make sense to do the surrogate modeling of $f(x)$used by all CGO solvers.

f (*x*) is assumed to be a costly function while *c*(*x*) is assumed to be cheaply computed. Any costly constraints can be treated by adding penalty terms to the objective function in the following way:

$$\min_{x} \quad p(x) = f(x) + \sum_{j} w_j \max\left(0,\ c^j(x) - c_U^j,\quad c_L^j - c^j(x)\right),$$

where weighting parameters w_j have been added. The user then returns *p*(*x*) instead of *f* (*x*) to the CGO solver.

Calling Syntax

```
Result = arbfMIP(Prob,varargin)
Result = tomRun('arbfMIP', Prob);
```

Description of Inputs

Problem description structure. The following fields are used:

Input*NameFUNCS.fFUNCS.cx_Lx_Ub_Ub_LAc_Lc_UWarmStartMaxCPUuserPriLevOptPriLevSubf_LowoptParamMaxFuncIterPr*
Name of the problem. Used for security when doing warm starts. Name of function to compute the objective function. Name of function to compute the nonlinear constraint vector. Lower bounds on the variables. Must be finite. Upper bounds on the variables. Must be finite. Upper bounds for the linear constraints. Lower bounds for the linear constraints. Linear constraint matrix. Lower bounds for the nonlinear constraints. Upper bounds for the nonlinear constraints. Set true (non-zero) to load data from previous run from *cgoSave.mat* and re- sume optimization from where the last run ended. If *Prob.CGO.WarmStartInfo* has been defined through a call to *WarmDefGLOBAL*, this field is used instead of the *cgoSave.mat* file. All CGO solvers uses the same mat-file and structure field and can read the output of one another.Maximal CPU Time (in seconds) to be used. User field used to send information to low-level functions. Print Level. 0 = silent. 1 = Summary 2 = Printing each iteration. 3 = Info about local / global solution. 4 = Progress in x. Print Level in subproblem solvers, see help in *snSolve* and *gnSolve*.Lower bound on the optimal function value. If defined, used to restrict the target values into interval \[f Low,min(surface)\]. Structure with optimization parameters. The following fields are used: Maximal number of costly function evaluations, default 300 for *rbfSolve* and *arbfMIP*, and default 200 for *ego*. *MaxFunc* must be <= 5000. If WarmStart = 1 and MaxFunc <= nFunc (Number of f(x) used) then set MaxFunc := MaxFunc + nFunc.Print one information line each iteration, and the new x tried. Default IterPrint = 1. fMinI means the best f(x) is infeasible. fMinF means the best f(x) is feasible (also integer feasible). Goal for function value, not used if *inf* or empty.Relative accuracy for function value, *fTol* == *eps_f* . Stop if |*f* - *fGoal*| = |*f_Goal*| * *fTol* , if *fGoal* = 0. Stop if |*f* - *fGoal*| = *fTol* , if *fGoal* = 0. See the output field maxTri. Linear constraint tolerance. Nonlinear constraint tolerance. Maximal number of iterations used in the local optimization on the re- sponse surface in each step. Default 1000, except for pure IP problems, then max(GO.MaxFunc, MaxIter);.

Structure (*Prob.CGO*) with parameters concerning global optimization options.

The following general fields in Prob.CGO are used:

Type of strategy to get the initial sampled values:

Percent	Experimental Design	ExD
	Corner strategies	
900	All Corners	1
997	*xL* + *xU* + adjacent corners	2
998	*xU* + adjacent corners	3
999	*xL* + adjacent corners	4
	Deterministic Strategies	
0	User given initial points	5
94	DIRECT solver *glbFast*	6
95	DIRECT solver *glcFast*	6
96	DIRECT solver *glbSolve*	6
97	DIRECT solver *glcSolve*	6

98	DIRECT solver *glbDirect*	6
99	DIRECT solver *glcDirect*	6
	Latin Based Sampling	
1	Maximin LHD 1-norm	7
2	Maximin LHD 2-norm	8
3	Maximin LHD Inf-norm	9
4	Minimal Audze-Eglais	10
5	Minimax LHD (only 2 dim)	11
6	Latin Hypercube	12
7	Orthogonal Samling	13
	Random Strategies (pp in %)	
1pp	Circle surrounding	14
2pp	Ellipsoid surrounding	15
3pp	Rectangle surrounding	16

Negative values of Percent result in constrained versions of the experimental design methods 7-16. It means that all points sampled are feasible with respect to all given constraints.

For ExD 5,6-12,14-16 user defined points are used.

Number of sample points to be used in initial experimental design. *nSample* is used differently dependent on the value of Percent:

	(n)Sample:			
ExD	**< 0**	**= 0**	**> 0**	**[]**
1	2^d			
6	|n|			
7-11	d+ 1	d + 1	max (d + 1, n)	(d + 1)(d + 2)/2
12	LATIN(k)			
13	|n|			
14-16	d + 1			

where LATIN = [21 21 33 41 51 65 65] and k = |*nSample*|. Otherwise nSample as input does not matter.

Description of the experimental designs:

ExD 1, All Corners. Initial points is the corner points of the box given by Prob.x L and Prob.x U. Generates 2d points, which results in too many points when the dimension is high.

ExD 2, Lower and Upper Corner point + adjacent points. Initial points are

2 * d + 2 corners: the lower left corner xL and its d adjacent corners xL + (xU (i) - xL (i)) * ei , i = 1, ..., d and the upper right corner xU and its d adjacent corners xU - (xU (i) - xL (i)) * ei , i = 1, ..., d

ExD 3. Initial points are the upper right corner xU and its d adjacent corners xU - (xU (i) - xL (i)) * ei , i = 1, ..., d

ExD 4. Initial points are the lower left corner xL and its d adjacent corners xL + (xU (i) - xL (i)) * ei , i = 1, ..., d

ExD 5. User given initial points, given as a matrix in CGO.X. Each column is one sampled point. If d = length(Prob.x L), then size(X,1) = d, size(X,2) = d + 1. CGO.F should be defined as empty, or contain a vector of corresponding f (x) values. Any CGO.F value set as NaN will be computed by solver routine.

ExD 6. Use determinstic global optimization methods to find the initial design. Current methods available (all DIRECT methods), dependent on the value of Percent:

99 = glcDirect, 98 = glbDirect, 97 = glcSolve, 96 = glbSolve, 95 = glcFast, 94 = glbFast.

ExD 7-11. Optimal Latin Hypercube Designs (LHD) with respect to different norms. The following norms and designs are available, dependent on the value of Percent:

1 = Maximin 1-Norm, 2 = Maximin 2-Norm, 3 = Maximin Inf-Norm, 4 = Audze-Eglais Norm, 5 = Minimax 2-Norm.

All designs taken from: http://www.spacefillingdesigns.nl/ [1]

Constrained versions will try bigger and bigger designs up to M = max(10 * *d, nT rial*) different designs, stopping when it has found nSample feasible points.

ExD 12. Latin hypercube space-filling design. For nSample < 0, k = |*nSample*| should in principle be the problem dimension. The number of points

sampled is:

k : 2 3 4 5 6 > 6

Points : 21 33 41 51 65 65

The call made is: X = daceInit(abs(nSample),Prob.x L,Prob.x U); Set nSample = [ ] to get (d+1)*(d+2)/2 sampled points:

d : 1 2 3 4 5 6 7 8 9 10

Points : 3 6 10 15 21 28 36 45 55 66

This is a more efficient number of points to use.

If CGO.X is nonempty, these points are verified as in ExD 5, and treated as already sampled points. Then nSample additional points are sampled, restricted to be close to the given points.

Constrained version of Latin hypercube only keep points that fulfill the linear and nonlinear constraints. The algorithm will try up to M = *max*(10 * *d, nTrial*) points, stopping when it has found nSample feasible points (d + 1 points if *nSample* < 0).

ExD 13. Orthogonal Sampling, LH with subspace density demands.

ExD 14-16. Random strategies, the |*Percent*| value gives the percentage size of an ellipsoid, circle or rectangle around the so far sampled points that new points are not allowed in. Range 1%-50%. Recommended values 10% - 20%.

If CGO.X is nonempty, these points are verified as in ExD 5, and treated as already sampled points. Then nSample additional points are sampled, restricted to be close to the given points.

The fields X,F,CX are used to define user given points. ExD = 5 (Percent = 0) needs this information. If ExD == 6-12,14-16 these points are included into the design. A matrix of initial x values. One column for every x value. If ExD == 5, size(X,2) >= dim(x)+1 needed.A vector of initial f (x) values. If any element is set to NaN it will be computed.Optionally a matrix of nonlinear constraint c(x) values. If nonempty, then size(CX,2) == size(X,2). If any element is set as NaN, the vector c(x) = CX(:,i) will be recomputed. If >= 0, *rand*(*'state', RandState*) is set to initialize the pseudo-random generator. If < 0, *rand*(*'state',* 100 * *clock*) is set to give a new set of random values each run. If isnan(RandState), the random state is not initialized. RandState will influence if a stochastic initial experimental design is applied, see input Percent and nSample. RandState will also influence if using the *multiMin* solver, but the random state seed is not reset in *multiMin*. The state of the random generator is saved in the warm start output rngState, and the random generator is reinitialized with this state if warm start is used. Default RandState = 0.If = 1, add the midpoint as extra point in the corner strategies. Default 1 for any corner strategy, i.e. Percent is 900, 997, 998 or 999. For experimental design CLH, the method generates M = *max*(10 * *d, nTrial*) trial points, and

evaluate them until *nSample* feasible points are found. In the random designs, *nTrial* is the maximum number of trial points randomly generated for each new point to sample.

Different search strategies for finding feasible LH points. First of all, the least infeasible point is added. Then the linear feasible points are considered. If more points are needed still, the nonlinear infeasible points are added.

1 - Take the sampled infeasible points in order.

2 - Take a random sample of the infeasible points.

3 - Use points with lowest constraint error (cErr).

0 - Original search space (default if any integer values).

1 - Transform search space to unit cube (default if no integers).

0 - No replacement, default for constrained problems.

1 - Large function values are replaced by the median.

> 1 - Large values Z are replaced by new values. The replacement is defined as $Z := FMAX + log10(Z - FMAX + 1)$, where $FMAX = 10^{REPLACE}$, if $min(F) < 0$ and $FMAX = 10^{(ceil(log10(min(F)))+REPLACE)}$, if $min(F) >= 0$. A new replacement is computed in every iteration, because $min(F)$ may change. Default REPLACE = 5, if no linear or nonlinear constraints.

0 - No local searches after global search. If RBF surface is inaccurate, might be an advantage.

1 - Local search from best points after global search. If equal best function values, up to 20 local searches are done.

1 - The problem is smooth enough for local search using numerical gradient estimation methods (default).

0 - The problem is nonsmooth or noisy, and local search methods using numer- ical gradient estimation are likely to produce garbage search directions.

Global optimization solver used for subproblem optimization. Default *glcCluster* (SMOOTH=1) or *glcDirect* (SMOOTH=0). If the global Solver is *glcCluster*, the fields *Prob.GO.maxFunc1*, *Prob.GO.maxFunc2*, *Prob.GO.maxFunc3*, *Prob.GO.localSolver*, *Prob.GO.DIRECT* and other fields set in *Prob.GO* are used. See the help for these parameters in *glcCluster*.

Local optimization solver used for subproblem optimization. If not defined, the TOMLAB default constrained NLP solver is used.

- Special RBF algorithm parameters in Prob.CGO -

Type of radial basis function: 1 - thin plate spline; 2 - Cubic Spline (default); 3 - Multiquadric; 4 - Inverse multiquadric; 5 - Gaussian; 6 - Linear. Global search type, always idea = 1, i.e. use fnStar values. if *fStarRule* =3, then N=1 default, otherwise N=4 default. By default *idea* =1, *fStarRule* =1, i.e. N =4. To change N, see below.Cycle length in idea 1 (default N=1 for fStarRule 3, otherwise default N=4) or idea 2 (always N=3).

If =1, add search step with target value *-8* first in cycle. Default 0. Always

=1 for the case *idea* =1, *fStarRule* =3.

Global-Local search strategy in idea 1, where N is the cycle length. Define *minsn* as the global minimum on the RBF surface. The following strategies for setting the target value *fStar* is defined: 1: $fStar = min_{sn} - ((N - (n - nInit))/N)^2 * \Delta n$ (Default), 2: $fStar = min_{sn} - (N - (n - nInit))/N * \Delta n$.

Strategy 1 and 2 depends on Δ_n estimate (see DeltaRule). If *infStep* =1, add $-\infty$-step first in cycle. 3: fStar = $-\infty$ -step, min_{sn}-k *0.1*$|min_{sn}|k = N, ..., 0$.

These strategies had the following names in Gutmanns thesis: III, II, I.

1 = Skip large f(x) when computing f(x) interval δ. 0 = Use all points. Default 1. Relative tolerance used to test if the minimum of the RBF surface, *minsn* , is sufficiently lower than the best point (*fM in*) found (default is 10-7).Max number of cycles without progress before stopping, default 10. Structure *Prob.GO* (Default values are set for all fields).Maximal number of function evaluations in each global search. Maximal number of iterations in each global

search. DIRECT solver used in glcCluster, either glcSolve or glcDirect(default). glcCluster parameter, maximum number of function evaluations in the first call. Only used if globalSolver is glcCluster, see help globalSolver. glcCluster parameter, maximum number of function evaluations in the second call. Only used if globalSolver is glcCluster, see help globalSolver.

glcCluster parameter, maximum sum of function evaluations in repeated first calls to DIRECT routine when trying to get feasible. Only used if globalSolver is glcCluster, see help *globalSolver*. *localSolver* The local solver used by glcCluster. If not defined, then *Prob.CGO.localSolver* is used *MIP* Structure in Prob, Prob.MIP.

Defines integer optimization parameters. Fields used:

If empty, all variables are assumed non-integer.

If islogical(IntVars) (=all elements are 0/1), then 1 = integer variable, 0 = continuous variable. If any element > 1, IntVars is the indices for integer variables.

Other parameters directly sent to low level routines.

Description of Outputs

Structure with result from optimization. The following fields are changed:

Output	Description
x_k	Matrix with the best points as columns.
f_k	The best function value found so far.
Iter	Number of iterations.
FuncEv	Number of function evaluations.
ExitText	Text string with information about the run.
ExitFlag	Always 0.
CGO	Subfield *WarmStartInfo* saves warm start information, the same information as in cgoSave.mat, see below.
Inform	Information parameter. 0 = Normal termination. 1 = Function value f(x) is less than fGoal. 2 = Error in function value $f(x)$, $\|f - fGoal\| = fTol$, $fGoal = 0$. 3 = Relative Error in function value $f(x)$ is less than fTol, i.e. $\|f - fGoal\|/\|fGoal\| = fTol$. 4 = No new point sampled for MaxCycle iteration steps. 5 = All sample points same as the best point for MaxCycle last iterations. 6 = All sample points same as previous point for MaxCycle last iterations. 7 = All feasible integers tried. 9 = Max CPU Time reached.
cgoSave.mat	To make a warm start possible, all CGO solvers saves information in the file cgoSave.mat. The file is created independent of the solver, which enables the user to call any CGO solver using the warm start information. cgoSave.mat is a MATLAB mat-file saved to the current directory. If the parameter SAVE is 1, the CGO solver saves the mat file every iteration, which enables the user to break the run and restart using warm start from the current state. SAVE = 1 is currently always set by the CGO solvers. If the cgoSave.mat file fails to open for writing, the information is also available in the output field Result.CGO.WarmStartInfo, if the run was concluded without interruption. Through a call to WarmDefGLOBAL, the Prob structure can be setup for warm start. In this case, the CGO solver will not load the data from cgoSave.mat. The file contains the following variables:
Name	Problem name. Checked against the *Prob.Name* field if doing a warmstart.
O	Matrix with sampled points (in original space).
X	Matrix with sampled points (in unit space if SCALE==1)
F	Vector with function values (penalty added for costly Cc(x))

F m	Vector with function values (replaced).
F00	Vector of pure function values, before penalties.
Cc	MMatrix with costly constraint values, *C* $c(x)$. *nInit*Number of initial points.
Fpen	Vector with function values + additional penalty if infeasible using the linear constraints and noncostly nonlinear $c(x)$.
fMinIdx	Index of the best point found.
rngState	Current state of the random number generator used.

Description

arbfMIP implements the Adaptive Radial Basis Function (ARBF) algorithm. The ARBF method handles linear equality and inequality constraints, and nonlinear equality and inequality constraints, as well as mixed-integer problems.

M-files Used

daceInit.m, *iniSolve.m*, *endSolve.m*, *conAssign.m*, *glcAssign.m*, *snSolve.m*, *gnSolve.m*, *expDesign.m*.

MEX-files Used

tomsol

See Also

rbfSolve.m and *ego.m*

Warnings

Observe that when cancelling with CTRL+C during a run, some memory allocated by arbfMIP will not be deallocated. To deallocate, do:

```
''>> ''clear cgolib
```

CGO Solver Reference

This page is part of the CGO Manual. See CGO Manual.

A detailed description of the TOMLAB /CGO solvers is given below. Also see the M-file help for *rbfSolve.m*, *ego.m* and *arbfMIP.m*.

rbfSolve

- rbfSolve

ego

- ego

arbfMIP

- arbfMIP

CGO rbfSolve description

This page is part of the CGO Manual. See CGO Manual.

Following is a detailed description of the *rbfSolve* algorithm.

Summary

The manual considers global optimization of costly objective functions, i.e. the problem of finding the global minimum when there are several local minima and each function value takes considerable CPU time to compute. Such problems often arise in industrial and financial applications, where a function value could be a result of a time-consuming computer simulation or optimization. Derivatives are most often hard to obtain, and the algorithms presented make no use of such information.

The emphasis is on a new method by Gutmann and Powell, *A radial basis function method for global optimization*. This method is a response surface method, similar to the Efficient Global Optimization (EGO) method of Jones. The TOMLAB implementation of the Radial Basis Function (RBF) method is described in detail.

Introduction

The task of global optimization is to find the set of parameters x in the feasible region $\Omega \subset R^d$ for which the objective function $f(x)$ obtains its smallest value. In other words, a point x^* is a *global optimizer* to $f(x)$ on Ω, if $f(x^*) \leq f(x)$. On the other hand, a point $\hat{x}$ is a *local optimizer* to $f(x)$, if $f(x) = f(x)$ for all x in some neighborhood around x. Obviously, when the objective function has several local minima, there could be solutions that are locally optimal but not globally optimal and standard local optimization techniques are likely to get stuck before the global minimum is reached. Therefore, some kind of global search is needed to find the global minimum with some reliability.

Previously a Matlab implementations of the DIRECT has been made, the new constrained DIRECT and the Efficient Global Optimization (EGO) algorithms. The implementations are part of the TOMLAB optimization environment.

The implementation of the DIRECT algorithm is further discussed and analyzed in Bjorkman, Holmström. Since the objective functions in our applications often are expensive to compute, we have to focus on very efficient methods. At the IFIP TC7 Conference on System Modelling and Optimization in Cambridge 1999, Hans-Martin Gutmann presented his work on the RBF algorithm. The idea of the RBF algorithm is to use radial basis function interpolation to define a utility function (Powell). The next point, where the original objective function should be evaluated, is determined by optimizing on this utility function. The combination of our need for efficient global optimization software and the interesting ideas of Powell and Gutmann led to the development of an improved RBF algorithm implemented in Matlab.

The RBF Algorithm

Our RBF algorithm is based on the ideas presented by Gutmann, with some extensions and further development. The algorithm is implemented in the Matlab routine *rbfSolve*.

The RBF algorithm deals with problems of the form

$$\min_x \quad f(x)$$
$$\text{s.t.} \quad x_L \leq x \leq x_U ,$$

where $f \in R$and $x, x_L, x_U \in R^d$. We assume that no derivative information is available and that each function evaluation is very expensive. For example, the function value could be the result of a time-consuming experiment or computer simulation.

Description of the Algorithm

We now consider the question of choosing the next point where the objective function should be evaluated. The idea of the RBF algorithm is to use radial basis function interpolation and a measure of 'bumpiness' of a radial function, σ say. A target value $f_n^*$is chosen that is an estimate of the global minimum of f. For each $y \neq \{x_1, ..., x_n\}$there exists a radial basis function that satisfies the interpolation conditions

$$s_y(x_i) = f(x_i), \quad i = 1, ..., n,$$
$$s_y(y) = f_n^*$$

The next point x_{n+1} is calculated as the value of y in the feasible region that minimizes $\sigma(s_y)$. It turns out that the function $y \mapsto \sigma(s_y)$is much cheaper to compute than the original function.

Here, the radial basis function interpolant s_n has the form

$$s_n(x) = \sum_{i=1}^{n} \lambda_i \phi\left(\|x - x_i\|_2\right) + b^T x + a,$$

with $\lambda_1, \ldots, \lambda_n \in R$, $b \in R^d$, $a \in R$and $\phi$is either cubic with $\phi(r) = r^3$or the thin plate spline $\phi(r) = r^2 \log r$. Gutmann considers other choices of $\phi$and of the additional polynomial, but later concludes that the situation in the multiquadric and Gaussian cases is disappointing.

The unknown parameters λ_i, b and a are obtained as the solution of the system of linear equations

$$\begin{pmatrix} \Phi & P \\ P^T & 0 \end{pmatrix} \begin{pmatrix} \lambda \\ c \end{pmatrix} = \begin{pmatrix} F \\ 0 \end{pmatrix}$$

where $\Phi$is the $n \times n$ matrix with $\Phi_{ij} = \phi\left(\|x_i - x_j\|_2\right)$and

$$P = \begin{pmatrix} x_1^T & 1 \\ x_2^T & 1 \\ . & . \\ . & . \\ x_n^T & 1 \end{pmatrix}, \lambda = \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ . \\ . \\ \lambda_n \end{pmatrix}, c = \begin{pmatrix} b_1 \\ b_2 \\ . \\ . \\ b_d \\ a \end{pmatrix}, F = \begin{pmatrix} f(x_1) \\ f(x_2) \\ . \\ . \\ f(x_n) \end{pmatrix}.$$

$s_y$could be obtained accordingly, but there is no need to do that as one is only interested in $\sigma(s_y)$. Powell shows that if the rank of *P* is *d* + 1, then the matrix

$$\begin{pmatrix} \Phi & P \\ P^T & 0 \end{pmatrix}$$

is nonsingular and the linear system (4) has a unique solution.

For s_n it is

$$\sigma(s_y) = \sigma(s_n) + \mu_n(y)\,[s_n(y) - f_n^*]^2$$

$$\sigma(s_n) = \sum_{i=1}^{n} \lambda_i s_n(x_i).$$

Further, it is shown that $\sigma(s_y)$is

$$y \notin \{x_1, \ldots, x_n\}.$$

Thus minimizing $\sigma(s_y)$subject to constraints is equivalent to minimizing g_n defined as

$$g_n(y) = \mu_n(y)\,[s_n(y) - f_n^*]^2, \quad y \in \Omega \setminus \{x_1, \ldots, x_n\},$$

where $\mu_n(y)$is the coefficient corresponding to *y* of the Lagrangian function *L* that satisfies $L(x_i) = 0$, $i = 1, \ldots, n$and *L*(*y*) = 1. It can be computed as follows. F is extended to

$$\Phi_y = \begin{pmatrix} \Phi & \phi_y \\ \phi_y^T & 0 \end{pmatrix},$$

where $(\phi_y)_i = \phi(\|y - x_i\|_2), i = 1, \ldots, n$and *P* is extended to

$$P_y = \begin{pmatrix} P \\ y^T \;\; 1 \end{pmatrix}.$$

Then $\mu_n(y)$is the (*n* + 1)-th component of $v \in R^{n+d+2}$that solves the system

$$\begin{pmatrix} \Phi_y & P_y \\ P_y^T & 0 \end{pmatrix} v = \begin{pmatrix} 0_n \\ 1 \\ 0_{d+1} \end{pmatrix}.$$

We use the notation 0_n and $0_{d+1}$for column vectors with all entries equal to zero and with dimension *n* and (*d* + 1), respectively. The computation of $\mu_n(y)$is done for many different *y* when minimizing $g_n(y)$. This requires $O(n^3)$ operations if not exploiting the structure of $\Phi_y$and P_y. Hence it does not make sense to solve the full system each time. A better alternative is to factorize the interpolation matrix and update the factorization for each *y*. An algorithm that requires $O(n^2)$operations is described in #Factorizations and Updates.

When there are large differences between function values, the interpolant has a tendency to oscillate strongly. It might also happen that $\min s_n(y)$is much lower than the best known function value, which leads to a choice of f_n^* that overemphasizes global search. To handle these problems, large function values are in each iteration replaced by the median of all computed function values. Note that $\mu_n$and $g_n$are not defined at $x_1, \ldots, x_n$and

$$\lim_{y \to x_i} \mu_n(y) = \infty, \quad i = 1, \ldots, n.$$

This will cause problems when $\mu_n$is evaluated at a point close to one of the known points. The function $h_n(x)$ defined by

$$h_n(x) \;=\; \begin{cases} \frac{1}{g_n(x)}, & x \notin \{x_1, \ldots, x_n\} \\ 0, & x \in \{x_1, \ldots, x_n\} \end{cases}$$

is differentiable everywhere on Ω, and is thus a better choice as objective function. Instead of minimizing $g_n(y)$ one may minimize $-h_n(y)$. In our implementation we instead minimize $-\log(h_n(y))$. By this we avoid a flat minimum and numerical trouble when $h_n(y)$is very small.

The Choice of *f* *

For the value of f_n^* it should hold that

$$f_n^* \in \left[-\infty, \min_{y\in\Omega} s_n(y)\right].$$

The case $f_n^* = \min_{y\in\Omega} s_n(y)$ is only admissible if $\min_{y\in\Omega} s_n(y) < s_n(x_i)$, <mah>i=1, \ldots, n</math>. There are two special cases for the choice of f_n^*. In the case when $f_n^* = \min_{y\in\Omega} s_n(y)$, then minimizing is equivalent to

$$\min_{y\in\Omega} s_n(y).$$

In the case when $f_n^* = -\infty$, then minimizing is equivalent to

$$\min_{y\in\Omega\setminus\{x_1,\ldots,x_n\}} \mu_n(y).$$

So how should f_n^* be chosen? If $f_n^* = -\infty$, then the algorithm will choose the new point in an unexplored region, which is good from a global search point of view, but the objective function will not be exploited at all. If $f_n^* = \min_{y\in\Omega} s_n(y)$, the algorithm will show good local behaviour, but the global minimum might be missed. Therefore, there is a need for a mixture of values for f_n^* close to and far away from $\min_{y\in\Omega} s_n(y)$. Gutmann describes two different strategies for the choice of f_n^*.

The first strategy, denoted **idea 1**, is to perform a cycle of length *N* + 1 and choose f_n^* as

$$f_n^* = \min_{y\in\Omega} s_n(y) - W \cdot \left(\max_i f(x_i) - \min_{y\in\Omega} s_n(y)\right),$$

with

$$W = \left[\frac{(N - (n - n_{init}))\mathrm{mod}(N+1)}{N}\right]^2,$$

where n_{init} is the number of initial points. Here, *N* = 5 is fixed and $\max_i f(x_i)$ is not taken over all points, except for the first step of the cycle. In each of the subsequent steps the $n - n_{max}$ points with largest function value are removed (not considered) when taking the maximum. Hence the quantity $\max_i f(x_i)$ is decreasing until the cycle is over. Then all points are considered again and the cycle starts from the beginning. More formally, if $(n - n_{init})\mathrm{mod}(N+1) = 0$, $n_{max} = n$, otherwise

$$n_{max} = \max\{2, n_{max} - \mathrm{floor}((n - n_{init})/N)\}$$

The second strategy, denoted **idea 2**, is to consider f_n^* as the optimal value of

$$\begin{aligned} \min \quad & f^*(y) \\ \text{s.t.} \quad & \mu_n(y)\left[s_n(y) - f^*(y)\right]^2 \le \alpha_n^2 \\ & y \in \Omega, \end{aligned}$$

and then perform a cycle of length *N* + 1 on the choice of *an* . Here, *N* = 3 is fixed and

$$\begin{aligned} \alpha_n &= \tfrac{1}{2}\left(\max_i f(x_i) - \min_{y\in\Omega} s_n(y)\right), \quad n = n_0, n_0+1 \\ \alpha_{n_0+2} &= \min\left\{1, \tfrac{1}{2}\left(\max_i f(x_i) - \min_{y\in\Omega} s_n(y)\right)\right\} \\ \alpha_{n_0+3} &= 0, \end{aligned}$$

where n_0 is set to *n* at the beginning of each cycle. For this strategy, $\max_i f(x_i)$ is taken over all points in all parts of the cycle.

Note that for a fixed *y* the optimal $f^*(y)$ is the one for which

$$\mu_n(y)\left[s_n(y) - f^*(y)\right]^2 = \alpha_n^2.$$

Substituting this equality constraint into the objective simplifies the problem to the minimization of

$$f^*(y) = s_n(y) - \alpha_n/\sqrt{\mu_n(y)}.$$

Denoting the minimizer by y^*, and choosing $f_n^* = f^*(y^*)$, it is evident that y^*minimizes $\mu_n(y)\left[s_n(y) - f_n^*\right]^2$ and hence $g_n(y)$.

For both strategies (**idea 1** and **idea 2**), a check is performed when $(n - n_{init})\mathrm{mod}(N+1) = N$. This is the stage when a purely local search is performed, so it is important to make sure that the minimizer of *sn* is not one of the interpolation points or too close to one. The test used is

$$f_{min} - \min_{y\in\Omega} s_n(y) \leq 10^{-4}\max\{1, |f_{min}|\},$$

where $f_{min}$is the best function value found so far, i.e. $\min_i f(x_i)$, $i = 1, \ldots, n$. For the first strategy (**idea 1**), then

$$f_n^* = \min_{y\in\Omega} s_n(y) - 10^{-2}\max\{1, |f_{min}|\},$$

otherwise $f_n^*$is set to 0. For the second strategy (**idea 2**), then *an* (or more correctly α_{n_0+3}) is set

$$\alpha_{n_0+3} = \min\left\{1, \frac{1}{2}\left(\max_i f(x_i) - \min_{y\in\Omega} s_n(y)\right)\right\},$$

otherwise $\alpha_{n_0+3}$is set to 0.

Factorizations and Updates

In Powell, a factorization algorithm is presented for the solution. The algorithm makes use of the conditional definiteness of Φ, i.e. $\lambda^T\Phi\lambda > 0,\ \lambda \neq 0$and $P^T\lambda = 0$. If

$$P = \begin{pmatrix} Y & Z \end{pmatrix}\begin{pmatrix} R \\ 0 \end{pmatrix}$$

is the *QR* decomposition of *P* , then the columns of *Z* span the null space of P^T, and every $\lambda$with $P^T\lambda = 0$can be expressed as $\lambda = Zz$for some vector *z*. Thus the conditional positive definiteness implies that

$$z^T Z^T \Phi Z z > 0,\ z \in R^{n-d-1}\ \setminus\ \{0\}.$$

This shows that $Z^T\Phi Z$is positive definite, and thus its Cholesky factorization

$$Z^T\Phi Z = LL^T$$

exists. This property can be used to solve (4) as follows. Consider the interpolation condition $\Phi\lambda + Pc = F$in (4). Multiply from left by $Z^T$and replace $\lambda$by Zz. Because $Z^TP = 0$, the interpolation condition simplifies to

$$Z^T\Phi Z z = Z^T F.$$

Solving this system using the Cholesky factorization gives *z*. Then compute $\lambda = Zz$and solve

$$Pc = F - \Phi\lambda$$

for *c* using the *QR* decomposition of *P* as

$$Rc = Y^T(F - \Phi\lambda).$$

The same principle can be applied to solve (12) for a given *y* to get $\mu_n(y)$. In analogy to the discussion above, if the extended matrices $\Phi_y$and $P_y$in (10) and (11), respectively, are given, and if

$$Z_y^T P_y = 0$$

and

$$Z_y^T\Phi_y Z_y = L_y L_y^T$$

is the Cholesky factorization, then the vector

$$v = Z_y z(y)$$

yields $\mu_n(y) = v_{n+1}$, where *z*(*y*) solves

$$Z_y^T\Phi_y Z_y z = Z_y\begin{pmatrix} 0_n \\ 1 \end{pmatrix}.$$

The Cholesky factorization is the most expensive part of this procedure. It requires (n^3)operations. As $\mu_n(y)$must be computed for many different *y* this is inacceptable. However, if one knows the *QR* factors of *P* and the Cholesky factor of *Z T* FZ , the *QR* factorization of P_y and the new Cholesky factor L_y can be computed in $O(n^2)$operations. The new $\Phi(y)$is

$$\Phi_y = \begin{pmatrix} \Phi & \phi_y \\ \phi_y^T & 0 \end{pmatrix},$$

where $\phi_y)_i = \phi(\|y - x_i\|_2), i = 1, \ldots, n$. The new *P* (*y*) is

$$P_y = \begin{pmatrix} P \\ y^T \quad 1 \end{pmatrix}.$$

Compute the *QR* factorization of P_y, defined in (10). Given $P = QR$, the *QR* factorization of $P_y$may be written as

$$P_y = Q_y R_y = \begin{pmatrix} Q & 0 \\ 0 & 1 \end{pmatrix} H R_y,$$

where $H$is an orthogonal matrix obtained by *d* + 1 Givens rotations and for $i = d + 2, \ldots, n$the *i*-th column of *H* is the *i*-th unit vector. Denote $B = Q^T \Phi Q$. Using $\Phi_y$as defined in (10) consider the expanded *B* matrix

$$\begin{aligned} B_y &= Q_y^T \Phi_y Q_y = H^T \begin{pmatrix} Q^T & 0 \\ 0 & 1 \end{pmatrix} \Phi_y \begin{pmatrix} Q & 0 \\ 0 & 1 \end{pmatrix} H = \\ &= H^T \begin{pmatrix} B & Q^T \phi_y \\ \phi_y^T Q & 0 \end{pmatrix} H. \end{aligned}$$

Multiplications from the right and left with *H* affects only the first (*d* + 1) rows and columns and the last row and the last columns of the matrix in the middle. (Remember, *d* is the dimension of the problem). Hence

$$B_y = \begin{pmatrix} * & * & * \\ * & Z^T \Phi Z & v \\ * & v^T & \gamma \end{pmatrix},$$

where * denotes entries not important for the moment. From the form of B_y it follows that

$$Z_y^T \Phi Z_y = \begin{pmatrix} Z^T \Phi Z & v \\ v^T & \gamma \end{pmatrix}$$

holds. The Cholesky factorization of $Z^T \Phi Z$is already known. The new Cholesky $L_y$factor is found by solving the lower triangular system $Ll = v$for *l*, computing $\beta = \sqrt{\gamma - l^T l}$, and setting

$$L_y = \begin{pmatrix} L & 0 \\ l^T & \beta \end{pmatrix}.$$

It is easily seen that $L_y L_y^T = Z_y^T \Phi_y Z_y$because

$$\begin{aligned} L_y L_y^T &= \begin{pmatrix} L & 0 \\ l^T & \beta \end{pmatrix} \begin{pmatrix} L^T & l \\ 0 & \beta \end{pmatrix} = \begin{pmatrix} LL^T & Ll \\ l^T L & l^T l + \beta^2 \end{pmatrix} = \\ &= \begin{pmatrix} Z^T \Phi Z & v \\ v^T & \gamma \end{pmatrix} = Z_y^T \Phi_y Z_y. \end{aligned}$$

Note that in practice we do the following: First compute the factorization of *P* , i.e. $P_y = Q_y R_y$, using Givens rotations. Then, since we are only interested in *v* and γin (42), it is not necessary to compute the matrix B_y in (41). Setting *v* to the last column in $Q_y$and computing $\tilde{v} = \Phi_y^T \hat{v} = \Phi_y \hat{v}$($\Phi_y$is symmetric), gives *v* and $\gamma$by multiplying the last (*n* - *d*) columns in Q_y *by* $tildev$, *i.e.*

$$\begin{pmatrix} v \\ \gamma \end{pmatrix} = Q_{y \cdot i}^T \tilde{v}, \quad i = d + 2, \ldots, n + 1.$$

Using this algorithm, *v* and γ are computed using ((*n* + 1) + (*n* - *d*)) inner products instead of the two matrix multiplications in (41).

Note that the factorization algorithm is a normal 'null-space' method for solving an optimization problem involving linear equality constraints. The system of linear equations in (4) defines the necessary conditions for a stationary point to the unconstrained quadratic programming (QP) problem

$$\min_{\lambda,c} \quad \frac{1}{2}\lambda^T\Phi\lambda + \lambda^T(Pc - F).$$

Viewing *c* as Lagrange multipliers for the linear equalities in (4), (47) is equivalent to the QP problem in λ defined as

$$\min_{\lambda} \quad \frac{1}{2}\lambda^T\Phi\lambda - F^T\lambda \quad \text{subject to} \quad P^T\lambda = 0.$$

The first condition in the conditional positive definiteness definition is the same as saying that the reduced Hessian must be positive definite at the solution of the QP problem if that solution is to be unique.

The type of update procedure described above is suitable each time an optimal point *y* = *xn*+1 is added. However, when evaluating all candidates *y* an even more efficient algorithm can be formulated. What is needed is a black-box procedure to solve linear systems with a general right-hand side:

$$\begin{pmatrix} \Phi & P \\ P^T & 0 \end{pmatrix} \begin{pmatrix} \lambda \\ c \end{pmatrix} = \begin{pmatrix} g \\ r \end{pmatrix}.$$

Using the *QR*-factorization in (28) the steps

$$\begin{aligned} R^T v &= r, \\ Z^T\Phi Z w &= Z^T(g - \Phi Y v), \\ \lambda &= Y v + Z w, \\ Rc &= Y^T(g - \Phi\lambda) \end{aligned}$$

simplify when *r* = 0 as in (4), but all steps are useful for solving the extended system (49); see next.

For each of many vectors *y*, the extended system takes the form

$$\left(\begin{array}{cc|c} \Phi & \phi & P \\ \phi^T & 0 & p^T \\ \hline P^T & p & 0 \end{array}\right) \begin{pmatrix} \bar{\lambda} \\ \mu \\ \bar{c} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix},$$

where $p^T = (\, y^T \quad 1\,)$. This permutes to

$$\left(\begin{array}{cc|c} \Phi & P & \phi \\ P^T & 0 & p \\ \hline \phi^T & p^T & 0 \end{array}\right) \begin{pmatrix} \bar{\lambda} \\ \bar{c} \\ \mu \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},$$

which may be solved by block-LU factorization (also known as the Schur-complement method). It helps that most of the right-hand side is zero. The solution is given by the steps

$$\begin{pmatrix} \Phi & P \\ P^T & 0 \end{pmatrix} \begin{pmatrix} \hat{\lambda} \\ \hat{c} \end{pmatrix} = \begin{pmatrix} \phi \\ p \end{pmatrix},$$

$$\mu = -1/(\phi^T\hat{\lambda} + p^T\hat{c}),$$

$$\begin{pmatrix} \bar{\lambda} \\ \bar{c} \end{pmatrix} = -\mu \begin{pmatrix} \hat{\lambda} \\ \hat{c} \end{pmatrix}.$$

Thus, each *y* requires little more than solving for $(\hat{\lambda}, \hat{c})$ using the current factorizations (two operations each with *Q*, *R* and *L*). This is cheaper than updating the factors for each *y*, and should be reliable unless the matrix in (4) is nearly singular. The updating procedure is best numerically, and it is still needed once when the final $y = x_{n+1}$ is chosen.

A Compact Algorithm Description

Section #Description of the Algorithm-#Factorizations and Updates described all the elements of the RBF algorithm as implemented in our Matlab routine *rbfSolve*, but our discussion has covered several pages. We now summarize everything in a compact step-by-step description. Steps 2, 6 and 7 are different in **idea 1** and **idea 2**.

	idea 1	**idea 2**
1:	Choose *n* initial points *x*1 , . . . , *xn* (normally the 2*d* box corner points defined by the variable bounds). Compute *Fi* = *f* (*xi*), *i* = 1, 2, . . . , *n* and set *ninit* = *n*.	
2:	Start a cycle of length 6.	Start a cycle of length 4.
3:	If the maximum number of function evalua- tions reached, quit.	
4:	Compute the radial basis function interpolant *sn* by solving the system of linear equations (4).	
5:	Solve the minimization problem min *sn* (*y*). *y?*?	
6:	Compute *f* * in (18) corresponding to the current position in the cycle.	Compute *an* in (22) corresponding to the current position in the cycle.
7:	New point *xn*+1 is the value of *y* that mini- mizes *gn* (*y*) in (9).	New point x_{n+1} is the value of *y* that min-imizes *f* *(*y*) in (24).
8:	Compute *Fn*+1 = *f* (*xn*+1) and set *n* = *n* + 1.	
9:	If end of cycle, go to **2**. Otherwise go to **4**.	

Some Implementation Details

The first question that arise is how to choose the points $x_1, \ldots, x_{n_{init}}$to include in the initial set. We only consider box constrained problems, and choose the corners of the box as initial points, i.e. $n_{init} = 2^d$. Starting with other points is likely to lead to the corners during the iterations anyway. But as Gutmann suggests, having a "good" point beforehand, one can include it in the initial set.

The subproblem

$$\min_{y \in \Omega} \quad s_n(y) \ ,$$

is itself a problem which could have more than one local minima. To solve (51) (at least approximately), we start from the interpolation point with the least function value, i.e. $mathrmargminf(x_i),\ i = 1, \ldots, n,$ and perform a local search. In many cases this leads to the minimum of s_n . Of course, there is no guarantee that it does. We use analytical expressions for the derivatives of *sn* and perform the local optimization using *ucSolve* in TOMLAB running the inverse BFGS algorithm.

To minimize $g_n(y)$for the first strategy, or $f^*(y)$for the second strategy, we use our Matlab routine *glbSolve* implementing the DIRECT algorithm (see the TOMLAB manual). We run *glbSolve* for 500 function evaluations and choose x_{n+1} as the best point found by *glbSolve*. When $(n - n_{init}) \bmod (N + 1) = N$(when a purely local search is performed) and the minimizer of *sn* is not too close to any of the interpolation points, i.e. (25) is not true, *glbSolve* is not used to minimize $g_n(y)$or $f^*(y)$. Instead, we choose the minimizer of (51) as the new point x_{n+1}. The TOMLAB routine *AppRowQR* is used to update the *QR* decomposition.

Our experience so far with the RBF algorithm shows that for the second strategy (**idea2**), the minimum of (24) is very sensitive for the scaling of the box constraints. To overcome this problem we transform the search space to the

unit hypercube. This algorithm improvement is necessary to avoid rank deficiency in the interpolation matrix for the train design problem.

In our implementation it is possible to **restart** the optimization with the final status of all parameters from the previous run.

Article Sources and Contributors

CGO *Source*: http://tomwiki.com/index.php?oldid=1117 *Contributors*: Elias

CGO Using the Matlab Interface *Source*: http://tomwiki.com/index.php?oldid=2627 *Contributors*: Elias

CGO Setting Options *Source*: http://tomwiki.com/index.php?oldid=1110 *Contributors*: Elias

CGO Test Examples *Source*: http://tomwiki.com/index.php?oldid=1111 *Contributors*: Elias

CGO rbfSolve *Source*: http://tomwiki.com/index.php?oldid=2393 *Contributors*: Elias

CGO ego *Source*: http://tomwiki.com/index.php?oldid=2394 *Contributors*: Elias

CGO arbfMIP *Source*: http://tomwiki.com/index.php?oldid=2395 *Contributors*: Elias

CGO Solver Reference *Source*: http://tomwiki.com/index.php?oldid=1112 *Contributors*: Elias

CGO rbfSolve description *Source*: http://tomwiki.com/index.php?oldid=2396 *Contributors*: Elias