

10 November 2010

CGO: Cayuga Global Optimizer

For use with MATLAB

User's Guide

Cayuga Research

Version 1.0

CGO: Cayuga Global Optimizer

©2008-2010 Cayuga Research Associates, LLC. All Rights Reserved.

Contents

0. Preface	1
1.0 About CGO.....	2
1.1 Write your objective function	3
2.0 Installation of CGO for Windows Users.....	5
2.1 Installation of CGO for Unix (Linux) Users.....	5
3.0 Use of the simulated annealing methodology	7
3.1 Simulated annealing options in CGO:	7
4.0 Independent local minimizations in Phase 2 for problem: $\min\{f(x) : l \leq x \leq u\}$	11
4.1 Independent local minimizations in Phase 2 for problem: $\min\{\ F(x)\ _2^2 : l \leq x \leq u\}$	12
4.2 Independent local minimizations in Phase 2 for problem: $\min\{g^T x + \frac{1}{2} x^T H x : l \leq x \leq u\}$	12
5 Troubleshooting	13
6 Works Cited.....	15
<i>Appendix A</i> Derivation of formula in method SA+Q.....	16
<i>Appendix B</i> Applying CGO with C/Fortran objective functions	17
<i>Appendix C</i> Use approximated objective functions in phase 1 in CGO	19
<i>Appendix D</i> Function Calls in CGO	20
<i>Appendix E</i> Using the CGO Function Calls	24

0. Preface

Many problems in the sciences, engineering disciplines, economics and planning, medicine and health, and the communication industry can be expressed as global optimization problems. Indeed, the global optimization problem is often the ‘driver’, i.e., the overarching problem to be solved. For example, maximize profit, minimize risk, find the shortest path, determine the energy-minimizing ‘fold’ for a molecule, determine the ‘best’ design for a craft, subject to material and cost constraints, and so on. ‘Global’ indicates that the best solution is sought over a pre-defined region.

The Cayuga Global Optimizer (CGO) is a powerful tool, combining several technologies, to search for global optimizers for continuous optimization problems. CGO is designed for use in the MATLAB environment and has several outstanding features:

- Derivatives can be computed automatically using the Cayuga tool ADMAT: there is no need to supply derivative functions (or derivative approximation schemes).
- Phase 1 improves on a set of user-supplied starting points (or randomly-generated starting points) using one of several simulated-annealing methodologies (some variants require function evaluations only) and approximation schemes.
- Phase 2 employs Matlab local minimizers starting from the endpoints in Phase 1, to obtain a set of accurate local minimizers.
- Both Phase 1 and Phase 2 allow for easy use of parallelism: see Cayuga Research package PCGO.
- CGO allows for significant flexibility: the user can tailor a global optimization approach by using one of several options for Phase 1 (or replace the entire Phase 1) with a user-supplied procedure for improving on starting points; there is MATLAB-flexibility in choosing or tailoring the local minimization procedures in Phase 2.
- User supplied functions can be MATLAB M-files or C, Fortran functions wrapped as MATLAB Mex-files. However, the Cayuga package ADMAT can only be applied to MATLAB M-files.
- Either Phase 1 or Phase 2 of CGO can be employed independently.
- CGO is driven either using an easy-to-use graphical user interface (GUI), or Matlab function calls (see Appendix D for details on the latter).

1.0 About CGO

Most global optimization problems are very hard to solve in the sense of guaranteeing convergence to a global minimizer. If a problem is known to be strictly convex, thus implying there is only a single local solution, then methods that guarantee to find a local solution will, in principle, find the global (= local) solution. However, most practical problems are not convex and therefore methods that guarantee to find a local solution will do exactly that: find a local, but not global, solution. A local solution means that the best point in a (possibly very small) neighbourhood has been found, but generally nothing can be said about this point relative to the global solution or even relative to other local minimizers.

It is generally not feasible to guarantee, in a mathematical sense, to converge to a global optimizer to a general continuous optimization problem. Some methods, e.g., simulated annealing (SA), will guarantee a global solution in a statistical sense, requiring infinite time [e.g., 2]. CGO generally uses SA-type methods, with a time limit, to improve on the initial set of starting points. (For functions that are expensive to evaluate, CGO allows for the use of less-expensive approximate functions in the use of SA-type methods to improve on the initial set of starting points.) CGO follows this starting-point improvement phase with independent local minimization processes to determine a set of accurate local minimizers; the best such point is identified as the ‘global’ solution.

Currently CGO addresses the following global optimization problems:

$$\min \{ f(x) : l \leq x \leq u \}, \quad (1.1)$$

$$\min \left\{ \|F(x)\|_2^2 : l \leq x \leq u \right\}, \quad (1.2)$$

$$\min \left\{ g^T x + \frac{1}{2} x^T H x : l \leq x \leq u \right\}. \quad (1.3)$$

In (1.1) it is assumed that the function f maps n -vectors x to real numbers, and that f is a continuous function on the box $l \leq x \leq u$. The n -vectors l, u represent the lower and upper bounds respectively, $l < u$. If f is continuous but not twice continuously differentiable, then CGO has a set of approaches, based on the simulated annealing idea, to search for a global minimizer. On the other hand, if f is twice continuously differentiable on the feasible region then CGO presents a 2-phase approach. Phase 1 allows for a choice of methods based on the simulated annealing philosophy to improve on a set of starting points; Phase 2 executes a set of independent (local) minimizations based on the endpoints determined in Phase 1; the best of the solutions determined in Phase 2 is then identified as the global solution.

In problem (1.2) F is a vector-valued mapping that takes n -vectors x to m -vectors $F(x)$. The n -vectors l, u represent the lower and upper bounds respectively, $l < u$. If F is not differentiable then CGO searches for a global minimizer using simulated annealing methodology based on function values alone. On the other hand, if F is differentiable then CGO searches for a global minimizer using two distinct phases. Phase 1 allows for a choice of methods based on the simulated annealing philosophy to improve on a set of initial starting points; Phase 2 executes a set of independent least-squares (local) minimizations, with Matlab function 'lsqnonlin' [1], based on the endpoints determined in Phase 1; the best of the solutions determined in Phase 2 is then identified as the global minimizer.

In problem (1.3) it is assumed that H is a symmetric n -by- n indefinite matrix, g is an n -vector, and l, u are each n -vectors representing lower and upper bounds respectively, $l < u$. The CGO approach to (1.3) has two phases. Phase 1 starts with a set of initial starting points and improves on them via one of several simulated annealing processes (based on function evaluations); Phase 2 applies independent local minimizations (Matlab function 'quadprog') from each of the endpoints from Phase 1; The 'global solution' is then chosen as the best of these improved endpoints.

Phase 2 of CGO for problems (1.1) and (1.2) requires computation of derivatives. In addition, some variants of the simulated annealing approaches in Phase 1 also require derivatives. The user can avoid coding derivative functions, or using finite-differences, by connecting CGO to the automatic differentiation package ADMAT. This merely involves indicating this in the ADMAT box on the GUI driver (or setting the flag in the function call) and derivatives will be computed efficiently and accurately. Three CGO function calls, `cgobndmin`, `cgobndnls`, and `cgobndquad` are explained and illustrated in detail in Appendix D and E.

1.1 Write your objective function

The minimum requirement CGO imposes on the objective function is to use the form:

$$f = \text{myfun}(x, \text{varargin}),$$

where the minimization of f is over x . Additional requirements depend on the chosen method of solution and are outlined below. The MATLAB Optimization Toolbox is necessary to perform all algorithms in CGO.

Algorithm SA stands for simulated annealing; SA+Q, SA+num avg, and SA>true avg are variants of SA with some type of smoothing method. Other user-defined functions may be needed depending on the algorithm chosen. See the table below for details.

Possible Phase 1 Algorithms	Applicable to problems	Objective function format and other functions needed
SA	Available for all dimensions	$f = \text{myfun}(x, \text{varargin})$
SA+Q	Available for all dimensions	$[f, g, H] = \text{myfun}(x, \text{varargin})$ where g is the gradient and H is the Hessian. Or, $f = \text{myfun}(x, \text{varargin})$ and use ADMAT
SA+num avg	1D	$f = \text{myfun}(x, \text{varargin})$
	2D	Write the objective function in the form: $f = \text{myfunint}(x_1, x_2)$ Dot operations such as $.*$ and $./$ should be used in <code>myfunint.m</code> to calculate objective function f instead of $*$ and $/$.
SA>true avg	1D	$[f, g, H, \text{ltrue}] = \text{myfun}(x, \text{varargin})$ where $\text{ltrue} = \int f(x)dx$
	2D	Additional user-defined function needed besides objective function $f = \text{myfun}(x, \text{varargin})$: $\text{ftrueint} = \text{functionname trueint}(a, b, c, d, \text{varargin})$ where $\text{ftrueint} = \int_a^b \int_c^d f(x_1, x_2) dx_2 dx_1$

In phase 2, depending on how the objective function is defined, different methods will be used to calculate derivatives.

Objective function given	Use ADMAT	Methods used in phase 2 to calculate derivatives
$f = \text{myfun}(x, \text{varargin})$	NO	Finite differencing
	YES	ADMAT
$[f, g, H] = \text{myfun}(x, \text{varargin})$	NO	User-supplied derivatives
	YES	ADMAT

2.0 Installation of CGO for Windows Users

CGO is provided in a zip file CGO.zip. Unzip the file CGO.zip and obtain a folder named CGO that contains the CGO package. You may place CGO in a folder of your choice.

The steps for installing CGO are as follows:

1. Click “File” in the MATLAB window.
2. Choose the “Set Path” option.
3. Click the “Add with Subfolders” button.
4. Find the folder for CGO in the “Browse for Folders” window and click “OK”.
5. Click the “Save” button to save the paths for CGO and click the “Close” button.
6. Set the CGO folder to be the current folder of the MATLAB command window.
7. Type “startup” to activate CGO.
8. Type “Y” in the MATLAB command window to accept the end-user licensing agreement.

The message ‘CGO installed successfully’ should appear in the MATLAB command window if CGO is properly installed. The steps 1-8 described above need to be performed only once during the installation of CGO.

In order to use the full spectrum of functionalities offered by CGO, the **Automatic Differentiation Toolbox ADMAT 2.0** from Cayuga Research must be installed as well. Please consult the User’s Guide for ADMAT 2.0 for the appropriate installation of ADMAT 2.0.

2.1 Installation of CGO for Unix (Linux) Users

1. Unzip CGO.zip using “unzip CGO.zip” at the Unix (Linux) prompt.
2. Access the CGO directory.
3. Edit **startup.m** file. Add ALL subdirectories of CGO search paths in the file manually.
4. Save the file and type **startup** at the MATLAB prompt to set up the paths for CGO.
5. Type ‘Y’ in the command window of MATLAB to accept the end-user licensing agreement.

2.2 CGO Activation

After the proper installation of CGO, activation of CGO may be needed in a newly-launched MATLAB session.

If the message ‘CGO installed successfully’ appears in the MATLAB command window at the start of a new MATLAB session, no activation of CGO is needed. CGO is ready for use.

Otherwise, first make sure that CGO has been properly installed. Then make the folder for CGO

as the current folder of the MATLAB command window; type “startup” to activate CGO. The message ‘CGO installed successfully’ should appear in the MATLAB command window, which indicates that CGO is now activated and ready for use.

3.0 Use of the simulated annealing methodology

Phase 1, discussed above, involves use of simulated annealing (SA) technology. We note that some variants available in Phase 1 can be applied without a differentiability assumption. The simulated annealing methodology has a rich literature, [e.g., (2), (3), (4)], and is derived from the physical annealing process where temperature is slowly lowered with respect to a molecular structure. A high temperature means the molecules are unsettled and exhibit high-amplitude vibrations. As the temperature is lowered, molecules settle and vibrational behaviour weakens. The idea in simulated annealing for optimization is to carry forward the notion of temperature lowering: with a high temperature points with somewhat higher function values can be accepted as new trial points (lower-valued points are always acceptable); as the temperature is lowered, with time, the algorithm becomes increasingly conservative about accepting points with higher values. The allowance of higher function values enables escape from basins around local minimizers. The main parameter in a simulated annealing approach is the rate at which the temperature is lowered. For background on simulated annealing ideas, see (2), (3), (4).

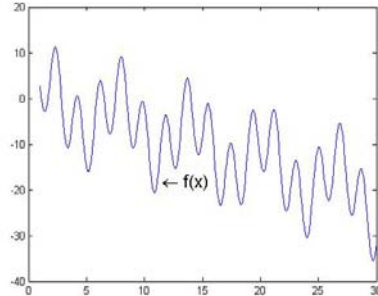
3.1 Simulated annealing options in CGO:

CGO allows for several choices in Phase 1 for improving on a set of starting points. These choices are variants on the simulated annealing idea. Below we discuss these variants and give some guidance as to how to choose the option most suitable for a given problem. Note that in Phase 1 it is sometimes advisable to replace the function to be minimized with a less-expensive approximation. We illustrate this in sample objective function ‘approximation problem 1’ in the third solver-CGO differentiable minimization problem solver and the objective functions in the last solver-CGO quadratic programming problem solver. See Appendix C for details about approximated objective function.

1. Choose ‘SA’ : This option triggers a ‘straight’ simulated annealing process. This option can be applied even if the objective function *is* not differentiable (but continuity is expected). The user can choose any number of different starting points (random uniformly distributed initial starting points are chosen by default) as well as the number of trials at each temperature setting. Different cooling schedules are also available. In theory, simulated annealing converges to a global minimum in a statistical sense (5) for continuous optimization problems though for most problems CGO uses SA as a procedure to improve starting points – local minimization techniques take over in Phase 2 and can generally obtain high quality local solutions more rapidly. An example problem in the ‘sample objective function’ set that is

particularly suitable is the “nesting” problem. See also the “1-d example 2” illustrated below.

$$\text{Example: } \min_x 6 \sin x + 8 \sin \frac{10x}{3} + \ln(x) - 0.84x, \quad 1 \leq x \leq 30$$

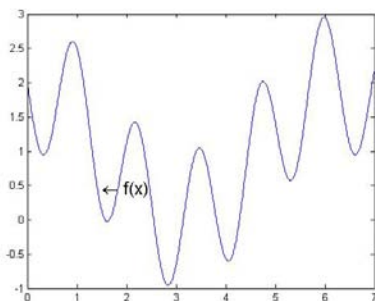


2. Choose ‘SA +Q’: Simulated annealing based on a local quadratic approximation to a smoothed function over the unit square. In this case the objective function must be twice continuously differentiable. Derivatives can be supplied ‘by hand’, or computed automatically using the automatic differentiator ADMAT. ‘SA + Q’ uses the simulated annealing framework but compares the ‘average values’, i.e., local quadratic approximation values, at each step. If there are many local minimizers with relatively high function values, this local quadratic approximation will eliminate some local minimizers because ‘average values’ within an interval are used. As temperature decreases, the interval length decreases to zero. In this way, the simulated annealing is applied to functions which increasingly approach the original function. Mathematically, the ‘averaged function value’ at x is defined as

$$\bar{f}(x) = f(x) + \frac{1}{6} \Delta \cdot \text{trace}(H)$$

where H is the Hessian of f at x . An example where ‘SA+Q’ is particularly good is the “1-d example 11” test problem.

$$\text{Example: } \min_x \cos x - \sin 5x + 1, \quad 0 \leq x \leq 7$$

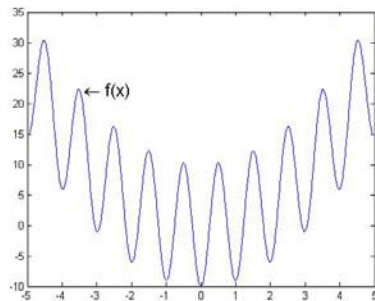


3. Choose ‘SA + num avg’: Similar to ‘SA+Q’ but the simulated annealing steps are performed with respect to a smoothed function that is computed using numerical methods for averaging over the unit square. In ‘SA+ num avg’, the ‘averaged values’ instead of true values are compared at each step of simulated annealing. The ‘averaged value’ of f in this method is defined over a rectangular Δ -box $Box(x)$, centered at x with sides $[x_i - \Delta, x_i + \Delta]$ as

$$\bar{f}(x) = \frac{1}{(2 * \Delta)^n} \int_{Box(x)} f(x) dx_1 dx_2 \cdots dx_n$$

In ‘SA+ num avg’, a numerical method is used to calculate the integration in the formula above. More specifically, adaptive Simpson quadrature is used to approximate the integration. This method is currently available to 1-D and 2-D problems only. An example that is quite amenable to this approach is “1-d example 3”.

Example: $\min_x x^2 - 10 \cos(2\pi x)$, $-5 \leq x \leq 5$



4. Choose ‘SA + true avg’: Simulated annealing based on the true average function value over the unit square around the current point. Note that this option is practical only for very low-dimensional problems (and special cases where the average is practical to compute). In ‘SA+ true avg’, the ‘average values’, i.e.,

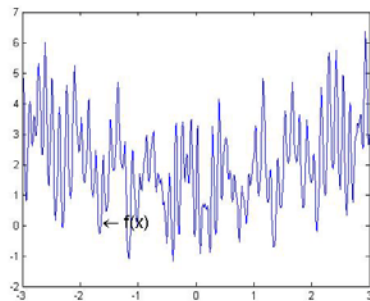
$$\frac{1}{(2 * \Delta)^n} \int_{Box(x)} f(x) dx_1 dx_2 \cdots dx_n$$

are compared at each step of the simulated annealing process. Close form expressions for the true integration are needed from the user. For example, in 1-D case the anti-derivative of $f(x)$ should be written as the fourth output in the objective function, i.e.,

[f, g, H, ltrue] = objfun(x, vargin). In 2-D case, a function of exact formula which evaluates the integration on a rectangular region should be provided, i.e., $f = \text{objfuntrueint}(x_1, y_1, x_2, y_2)$ where the inputs are the coordinates of a rectangular box.

When the closed form of integration is easy to obtain, this method can be quite effective. Below is a good example for this approach. To experiment, choose “1-d example 8” or “1-d example 9” from the list of sample objective functions.

Example: $\min_x \frac{x^2}{4} + e^{\sin 50x} + \sin(70 \sin x) - \sin(10x), -3 \leq x \leq 3$



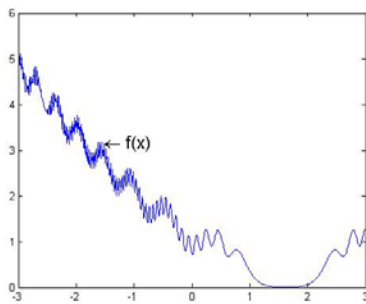
4.0 Independent local minimizations in Phase 2 for problem:

$$\min \{f(x) : l \leq x \leq u\}$$

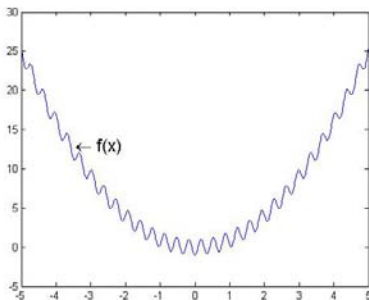
In CGO, local minimizations are performed by Matlab function 'fmincon'. Since gradients and Hessians are needed the user can either supply the derivative functions or, alternatively, turn 'on' ADMAT to compute first and second derivatives automatically. The following are good examples for experimentation.

Example: Try the "1-d example 9", "1-d example 19", or "Griewank".

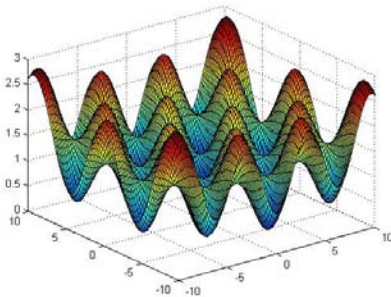
$$\min_x 0.2(x-1.62)^2 + 0.3\sin(0.5(x-1.62))^2 + 0.5\sin(1.1(x-1.62)^2)^2 + 0.4\sin(3(x-1.62)^3)^2, -3 \leq x \leq 3$$



Example: $\min_x x^2 - \cos(18x), -5 \leq x \leq 5$



Example: $\min_{x,y} 1 + \sin x^2 + \sin y^2 - 0.1e^{(-x^2-y^2)}, -10 \leq x \leq 10, -10 \leq y \leq 10$



4.1 Independent local minimizations in Phase 2 for problem:

$$\min \left\{ \|F(x)\|_2^2 : l \leq x \leq u \right\}$$

In CGO, local minimizations are performed by Matlab function 'lsqnonlin'. The user can explicitly supply the MATLAB code to evaluate Jacobian matrices or, alternatively, turn 'on' ADMAT to compute the Jacobian matrices automatically. The following are good examples for experimentation.

Example: Try the 1-D and 2-D "clusters problems", and "2d example in local volatility surface approximation".

4.2 Independent local minimizations in Phase 2 for problem:

$$\min \left\{ g^T x + \frac{1}{2} x^T H x : l \leq x \leq u \right\}$$

In CGO, local minimizations are performed by Matlab function 'quadprog'. The user supplies the vector g and the symmetric matrix H . The CGO quadratic problems solver has two sample sets of indefinite programming problems to try. The first set has vertex global solutions, and the second set exhibits global solutions on a more general boundary of the feasible region.

Example: See the test problems with the CGO quadratic problems solver.

5 Troubleshooting

Below we list some potential problems that may occur in the use of CGO.

1. ??? Error using ==> XXX
Too many input arguments.
Check the user-defined objective has two input arguments `x` and `varargin` in the following form: `f = myfun(x, varargin)`.
2. ??? Undefined function or method XXX for input arguments of type 'double'.
Some objective functions in the examples in CGO GUI are written in C. To compile a C program in MATLAB, it is usually required that a third-party compiler installed on your system. Please check <http://www.mathworks.com/support/compilers/R2010a/> for details. Once a compiler is installed, type `mex -setup` in MATLAB command window to select compiler configuration.

Below we list several potential problems that may occur in the use of ADMAT.

1. ??? Conversion to double from deriv is not possible.
This usually means a `deriv` class object is assigned to a double class variable. Check both sides of the assignment statement and make sure that they are of the same data type.
2. ??? Error using ==> XXX
Function XXX has not been defined for variables of class **deriv**. A number of MATLAB functions have not been overloaded in ADMAT yet. Please contact Cayuga Research for extending ADMAT to the MATLAB functions of your interest.
3. ??? Undefined function or variable **deriv**
ADMAT has not been installed yet. Please make sure that ADMAT is properly installed.
4. ??? Error using ==> deriv/derive
Please restart ADMAT. There may be a license problem.
ADMAT detects a possible license error. Please restart ADMAT.
5. The ADMAT 2.0 license has expired. Please contact Cayuga Research for a license extension.
This indicates that the license for ADMAT 2.0 has already expired. Please contact Cayuga Research for license renewal.

6. Do not use Matlab command **clear all** to clear your workspace while using ADMAT. This would remove all ADMAT global variables from memory: unpredictable errors may then occur. Instead, use **clear** selectively as needed.
7. ADMAT 2.0 only performs 1-D and 2-D matrix operations. In other words, it cannot perform 3-D or higher-dimension operations. use **clear** selectively as needed.
8. The computed derivatives are incorrect. Please check the following issues.
 - The command **clear all** should not be called while using ADMAT.
 - The data type of the dependent variable must be consistent with that of the input independent variable in a user-defined function (See Section 3.4 in the User's Guide for details).

Finally, if there is still an error, please contact Cayuga Research for help.

6 Works Cited

1. The MathWorks. *www.mathworks.com*. [Online]
2. [book auth.] Emile H. L. Aarts Peter J. M. Laarhoven. *Simulated annealing: theory and applications*. s.l. : Springer, 1987.
3. [book auth.] L. P. P. van Ginneken R. H. J. M. Otten. *The annealing algorithm*. s.l. : Kluwer Academic Publishers, 1989.
4. [book auth.] Paolo Sibani, Richard Frost Peter Salamon. *Facts, conjectures, and improvements for simulated annealing*. s.l. : Society for Industrial and Applied Mathematic , 2002.
5. *Convergence theorems for a class of simulated annealing algorithms on R^d* . **Belisle, C.J.P.** 4, 1992, Journal of Applied Probability, Vol. 29, pp. 885--895.

Appendix A

Derivation of the formula used in method SA+Q

We derive the formula $\bar{f}(x) = f(x) + \frac{1}{6}\Delta^2 \cdot \text{trace}(H)$ used in method SA+Q in Phase 1

Let f be an objective function and Δ be a positive number. The average value of f over a regular Δ -box $\text{Box}(x)$ centered at x with sides $[x_i - \Delta, x_i + \Delta]$ is

$$\bar{f}(x) = \frac{1}{(2 * \Delta)^n} \int_{\text{Box}(x)} f(x) dx_1 \cdots dx_n \quad (1.4)$$

The formula above is too expensive to compute when n is large or function f is difficult to compute. However, by approximating f using quadratic Taylor series expansion

$$f(x+s) \cong f(x) + g^T s + \frac{1}{2} s^T H s \equiv q(s)$$

where $g = \nabla f(x)$, $H = \nabla^2 f(x)$, (1.4) can be approximated as

$$\bar{f}(x) \cong \bar{q}(x) = f(x) + \frac{1}{(2 * \Delta)^n} \int_{\forall i, |s_i| \leq \Delta} (g^T s + \frac{1}{2} s^T H s) ds_1 \cdots ds_n$$

Since $g^T s + \frac{1}{2} s^T H s = \sum_i g_i s_i + \sum_i \sum_j s_i s_j h_{ij}$, interchanging the order of summation and

integration of the above formula yields

$$\bar{f}(x) = f(x) + \frac{1}{6}\Delta^2 \cdot \text{trace}(H)$$

Appendix B

Applying CGO with C/Fortran objective functions

Objective functions written in C/Fortran can be used as user-defined functions. Create a gateway function by using the mexFunction in your C/Fortran source file as the interface between your code and Matlab. Table 1 is an example written in C and saved as test5.c. f is the objective function value, g is gradient and H is Hessian. Use mex command to compile test5.c in Matlab to produce corresponding binary MEX-file.

Table 1 Example in C code

```
#include "mex.h"
#include "math.h"

void example2d(double f[], double g[],double H[], double x[])
{
    f[0] = 4*pow(x[0],2)-2.1*pow(x[0],4)+pow(x[0],6)/3+x[0]*x[1]-
4*pow(x[1],2)+4*pow(x[1],4);
    g[0] = 2*pow(x[0],5) - (42*pow(x[0],3))/5 + 8*x[0] + x[1];
    g[1] = 16*pow(x[1],3) - 8*x[1] + x[0];
    H[0] = 10*pow(x[0],4) - (126*pow(x[0],2))/5 + 8;
    H[1] = 1;
    H[2] = 1;
    H[3] = 48*pow(x[1],2) - 8;
}

void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
    double *f, *g, *H, *x;

    /* Create matrix for the return argument. */
    plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);
    plhs[1] = mxCreateDoubleMatrix(2,1, mxREAL);
    plhs[2] = mxCreateDoubleMatrix(2,2, mxREAL);
    /* Assign pointers to each input and output. */
    x = mxGetPr(prhs[0]);

    f = mxGetPr(plhs[0]);
    g = mxGetPr(plhs[1]);
    H = mxGetPr(plhs[2]);
    /* Call the timestwo subroutine. */
    example2d(f,g,H,x);
}
```

Once the MEX-files are obtained, simply provide one more Matlab function as Table 2 illustrates. Depending on the C/Fortran program and the algorithm chosen, g and H are optional. This trivial Matlab function is necessary for the Solvers to check for possible errors before the algorithm proceeds.

Table 2 Matlab objective function

```
function [f,g,H]=testmex2d(x,varargin)

[f,g,H]=test5(x);
```

Save the above Matlab code as testmex2d.m. Enter testmex2d as the name of user-defined function. Enter 2 as number of variables in this case. Define bounds and then click Solve.

Note that ADMAT can differentiate any function defined in an M-file. ADMAT cannot be applied to any external files, such as MEX files. So ADMAT cannot be used to get gradient and Hessian of a mex file or an M-file that calls a mex file.

Appendix C

Use approximated objective functions in phase 1 in CGO

Phase 1 seeks a good set of starting points for phase 2 (local minimization). When the evaluation of objective function is considered too expensive, consider using an approximated objective function in phase 1 with relatively cheap cost. In phase 2, the true objective function is used to obtain final results. Two examples of using approximated objective functions in phase 1 are given in two solvers in the CGO package: CGO differentiable minimization problem solver and CGO quadratic programming problem solver.

Example 1 (differentiable minimization): choose the 'approximation problem 1'

The true objective function is defined as

$$f(x) = z^T H_1 z + \gamma \cdot x^T H_2 x \text{ where vector } z \text{ is obtained from}$$

$$y_1 - \widehat{F}(x) = 0$$

$$A y_2 - y_1 = 0$$

$$z - \overline{F}(x) = 0$$

In phase 1 when evaluating function values, one can approximately solve $A y_2 - y_1 = 0$ by using the conjugate gradient method with a low tolerance (e.g., tol = 0.01).

Example 2 (Quadratic problem):

When 'use approximation in phase 1' is checked, the matrix H in the true objective function

$f = \frac{1}{2} x^T H x + g^T x$ will be approximated by

$$\widetilde{H} = \sum_{i=1}^k \lambda_i v_i v_i^T + \sum_{j=1}^k \lambda_j v_j v_j^T \quad (k = 2),$$

where the summations are on k most positive eigenvalues and k most negative eigenvalues and the corresponding eigenvectors.

Appendix D

Function Calls in CGO

Three function calls are available in CGO, i.e., *cgobndmin* for bounds constrained minimization problem, *cgobndnls* for bounds constrained nonlinear least squares problem and *cgobndquad* for bounds constrained quadratic programming. Detailed function usage is below.

```
function [xbest, fbest, elapsedtime] = cgobndmin(fun, x0 ,lb, ub,
phaseoptions, phase2options, otheroptions, varargin)

% cgobndmin seeks a global minimum of a bound constrained function of
% several variables. cgobndmin addresses problems of the form:
%
% min F(X) subject to: LB <= X <= UB (bounds)
% X
%
% X = cgobndmin(FUN, X0 ,LB, UB) starts at X0 and finds a minimum X to
% the function FUN, subject to the bounds constraint LB <= X <= UB. FUN
% accepts input X and returns a scalar function value F evaluated at X.
% X0 may be a scalar, vector, or matrix.
%
% X = cgobndmin(FUN, X0 ,LB, UB, phaseoptions, phase2options,
% otheroptions) minimizes with optimization parameters set by values
% in the structures phaseoptions, phase2options, otheroptions. Arguments
% are set with OPTIMSETPHASE1, OPTIMSET, and OPTIMSETOTHER. See
% OPTIMSETPHASE1, OPTIMSET, and OPTIMSETOTHER for details. For a list of
% options accepted by FMINCON refer to the OPTIMSET documentation.
%
% [X,FVAL] = cgobndmin(FUN,X0,...) returns the value of the objective
% function FUN at the solution X.
%
% [X,FVAL,elapsedtime] = cgobndmin(FUN,X0,...) returns an elapsed time
% in the solver.
%
% Examples
% FUN can be specified using function handle @:
% X = cgobndmin(@humps,...)
% In this case, F = humps(X) returns the scalar function value F of
% the HUMPS function evaluated at X.
%
% If FUN is parameterized, anonymous functions can be used to capture the
% problem-dependent parameters. Suppose the objective function is given
% in myfun, subject to the nonlinear constraint mycon, where these two
% functions are parameterized by their second argument a1 and a2,
% respectively. Here myfun is M-file function such as
%
% function f = myfun(x,a1)
% f = x(1)^2 + a1*x(2)^2;
%
% To optimize for a specific value of a1, first assign the value to the
```

```

% parameter. Next, create a one-argument anonymous function that captures
% the value of a1, and call myfun with a1. Finally, pass these anonymous
% functions to cgobndmin:
%
%     a1 = 2; % define parameter first
%     [xbest, fbest, elapsedtime] =
%     cgobndmin(@(x)testmyfuncall(x,a1), X0....
%
% See also OPTIMSET, optimsetphase1, optimset, optimsetother, @,
% FUNCTION_HANDLE.
%
% phase 1 options use default values
% T = 10*0.9.^(0:199);
% defaultopt = struct( ...
%     'TrialeachTem',5, ...
%     'SAalgorithm','SA', ...
%     'CoolingSchedule',T, ...
%     'SmoothingSchedule',linspace(1,0,200), ...
%     'UseApproximation','off', ...
%     'ApproximationFcn',[])
%
% phase 2 options default is the the same as default values of fmincon.
%
% other options have default values as below
% defaultoptother = struct( ...
%     'SaveDir', 'C:\mysavedmatfiles', ...
%     'NumofSoltoSave', 20, ...
%     'UseAdmat', 'off', ...
%     'MaxTimeinPhaseOne',[], ...
%     'Phase1checked', 'on', ...
%     'Phase2checked', 'on' ...
%     );

```

```

function [xbest, fbest, elapsedtime] = cgobndnls(fun, x0 ,lb, ub,
phaseloptions, phase2options, otheroptions, varargin)

% cgobndnls seeks a global minimum of a bound constrained nonlinear least
% squares problem of several variables. cgobndnls addresses problems of
% the form:
%
%     min ||F(X)||^2_2 subject to:  LB <= X <= UB      (bounds)
%     X
%
% X = cgobndnls(FUN, X0 ,LB, UB) starts at X0 and finds a minimum X to
% the function FUN, subject to the bounds constraint LB <= X <= UB. FUN
% accepts input X and returns a vector function value F evaluated at X.
% X0 may be a scalar, vector, or matrix.
%
% X = cgobndnls((FUN, X0 ,LB, UB, phaseloptions, phase2options,
% otheroptions) minimizes with optimization parameters given in the
% structures phaseloptions, phase2options, otheroptions. Arguments are

```

```

% set with OPTIMSETPHASE1, OPTIMSET, and OPTIMSETOTHER function. See
% OPTIMSETPHASE1, OPTIMSET, and OPTIMSETOTHER for details. For a list of
% options accepted by lsqnonlin refer to the OPTIMSET documentation.
%
% [X,FVAL] = cgobndnls(FUN,X0,...) returns the value of the objective
% function FUN at the solution X.
%
% [X,FVAL,elapsedtime] = cgobndnls(FUN,X0,...) returns an elapsed time
% in the solver.
%
% Examples
% FUN can be specified using function handle @:
%     X = cgobndnls(@myfun,...)
%     In this case, F = myfun(X) returns the vector function value F of
%     the myfun function evaluated at X.
%
% If FUN is parameterized, anonymous functions can be used to capture the
% problem-dependent parameters. Suppose the non-linear least squares
% problem is given in the function myfun, which is parameterized by its
% second argument c. Here myfun is an M-file function such as
%
%     function F = myfun(x,c)
%     F = [ 2*x(1) - exp(c*x(1))
%          -x(1) - exp(c*x(2))
%          x(1) - x(2) ];
%
% To solve the least squares problem for a specific value of c, first
% assign the value to c. Next, create a one-argument anonymous function
% that captures that value of c and calls myfun with two arguments.
% Finally, pass this anonymous function to LSQNONLIN:
%
%     c = -1; % define parameter first
%     x = cgobndnls(@(x) myfun(x,c),[1;1],...
%
% See also OPTIMSET, OPTIMSETPHASE1, OPTIMSETOTHER, @,
% FUNCTION_HANDLE.
%
% phase 1 options use default values
% T = 10*0.9.^(0:199);
% defaultopt = struct( ...
%     'TrialeachTem',5, ...
%     'SAalgorithm','SA', ...
%     'CoolingSchedule',T, ...
%     'UseApproximation','off', ...
%     'ApproximationFcn',[])
%
% phase 2 options default is the the same as default values of
% lsqnonlin.
%
% other options have default values as below
% defaultoptother = struct( ...
%     'SaveDir', 'C:\mysavedmatfiles', ...
%     'NumofSoltoSave', 20, ...
%     'UseAdmat', 'off', ...
%     'MaxTimeinPhaseOne',[], ...
%     'Phaselchecked', 'on', ...
%     'Phase2checked', 'on' );

```

```

function [xbest, fbest, elapsedtime] = cgobndquad(quad_H, quad_g, lb, ub, x0,
phaseoptions, phase2options, otheroptions, varargin)

% cgobndquad seeks a global minimum of a bound constrained quadratic
% function of several variables. cgobndquad addresses problems of
% the form:
%
% min quad_g'*X + 1/2*X'*quad_H*X subject to: LB <= X <= UB (bounds)
% X
%
% X = cgobndquad(quad_H, quad_g, LB, UB, X0) starts at X0 and seeks a
% minimum X to the quadratic quad_g'*X + 1/2*X'*quad_H*X, subject to the
bounds
% constraint LB <= X <= UB. X0 may be a scalar, vector, or matrix.
%
% X = cgobndquad(quad_H, quad_g, LB, UB, X0, phaseoptions,
% phase2options, otheroptions) minimizes with optimization parameters
% given in the structures phaseoptions, phase2options, otheroptions.
% Arguments are set with OPTIMSETPHASE1, OPTIMSET, and OPTIMSETOTHER. See
% OPTIMSETPHASE1, OPTIMSET, and OPTIMSETOTHER for details. For a list of
% options accepted by quadprog refer to the OPTIMSET documentation.
%
% [X,FVAL] = cgobndquad(quad_H, quad_g, LB, UB, X0...) returns the value
% of the objective function fun at the solution X.
%
% [X,FVAL,elapsedtime] = cgobndquad(quad_H, quad_g, LB, UB, X0...)
% returns an elapsed time in the solver.
%
% quad_g is a column vector; quad_H is a symmetric matrix. if quad_H is
% convex, solver will end after one starting point and output the global
% optimum.
%
% See also OPTIMSET, OPTIMSETPHASE1, OPTIMSETOTHER, @, FUNCTION_HANDLE.
%
% phase 1 options use default values
% T = 10*0.9.^(0:199);
% defaultopt = struct( ...
% 'TrialeachTem',5, ...
% 'SAalgorithm','SA', ...
% 'CoolingSchedule',T, ...
% 'SmoothingSchedule',linspace(1,0,200), ...
% 'UseApproximation','off', ...
% 'numofeig', 2}
%
% phase 2 options default is the the same as default values of quadprog.
%
% other options have default values as below
% defaulttoptother = struct( ...
% 'SaveDir', 'C:\mysavedmatfiles', ...
% 'NumofSoltoSave', 20, ...
% 'UseAdmat', 'off', ...
% 'MaxTimeinPhaseOne',[], ...
% 'Phase1checked', 'on', ...
% 'Phase2checked', 'on' };

```

Appendix E

Using the CGO Function Calls

CGO function calls `cgobndmin`, `cgobndnls`, and `cgobndquad` are available to users in the style of MATLAB function `fmincon`. The syntax of each of these three functions is given below,

To solve $\min_{lb \leq x \leq ub} f(x)$:

`[xbest, fbest, elapsedtime] = cgobndmin(fun, x0, lb, ub, phase1options, phase2options, otheroptions, varargin)`

To solve $\min_{lb \leq x \leq ub} \|F(x)\|_2^2$:

`[xbest, fbest, elapsedtime] = cgobndnls(fun, x0, lb, ub, phase1options, phase2options, otheroptions, varargin)`

To solve $\min_{lb \leq x \leq ub} \frac{1}{2} x^T H x + g^T x$:

`[xbest, fbest, elapsedtime] = cgobndquad(quad_H, quad_g, lb, ub, x0, phase1options, phase2options, otheroptions, varargin)`

Set the options for phase 1, phase 2, (and other options) as indicated below. Available options for each function are listed and explained (see the CGO Users Guide for more details). Default settings of these options are in the column on the right.

Phase 1 options	TrialeachTem	Number of trials at each temperature level in simulated annealing algorithm (SA) –type method. A positive integer is required.	5
	SAalgorithm	Choice of SA variant. Choices include 'sa', 'saq', 'sanumavg', and 'satruavg'.	'sa'
	CoolingSchedule	A vector with entries decreasing to zero for the temperatures used in simulated annealing algorithm or its variant.	$10 * 0.9.^{(0:199)}$
	SmoothingSchedule	A vector with entries decreasing to zero for the smoothing coefficients used in the SA variants 'saq',	<code>linspace(1,0,200)</code>

	UseApproximation	'sanumavg', and 'satruavg' algorithm . Choose an approximate objective function in Phase 1 if 'UseApproximation' is set to 'on'.	'off'
	ApproximationFcn	If 'UseApproximation' is set to 'on', provide function handle of the approximating function in this field.	[]
	Numofeig	Approximating function is written for the indefinite quadratic programming problem. A positive integer is required. See the User Guide for more details.	2
Phase 2 options	In cgobndmin, cgobndnls, and cgobndquad phase 2 options are the same as the options in MATLAB functions fmincon, lsqnonlin, and quadprog respectively.		See MATLAB help for fmincon, lsqnonlin and quadprog for default options.
Other options	SaveDir	Results in .mat format will be saved in this directory.	'C:\mysavedmatfiles'
	NumofSoltoSave	A set of optimizers and their function values can be saved.	20
	UseAdmat	If UseAdmat is set to 'on', the CGO package ADMAT will be used to get derivatives.	'off'
	MaxTimeinPhaseOne	A time bound for the application of Phase 1 (simulated annealing). A positive real number is required.	[]
	Phase1checked	If Phase1checked is set to 'on', Phase 1 will be executed.	'on'
	Phase2checked	If Phase2checked is set to 'on', Phase 2 (independent local minimizations) will be conducted	'on'

Note that

- phase 1 option 'Numofeig' is only applicable in cgobndquad;
- phase 1 option 'SmoothingSchedule' is not applicable in cgobndnls;
- phase 1 option 'ApproximationFcn' and 'ApproximationFcn' are not applicable in cgobndquad.
- 'SmoothingSchedule' should always have the same length as 'CoolingSchedule'.

Examples. A detailed illustration of use of these three function calls can be found in demo_cgofunctioncall.m in the folder CGO \Demos. Here we provide two brief examples.

Example 1: Solve $\min_{-10 \leq x, y \leq 10} 1 + \frac{x^2}{200} + \frac{y^2}{200} - \frac{\cos x \cos y}{\sqrt{2}}$ using cgobndmin with default options.

The objective function is written in griewank.m as follows.

```
function [f, g, H] = griewank(x, varargin)

f = (1 + x(1)*x(1)/200 + x(2)*x(2)/200 - cos(x(1))*cos(x(2))/sqrt(2));
if (nargout >= 2)
    g(1,1) = x(1)/100 + sin(x(1))*cos(x(2))/sqrt(2);
    g(2,1) = x(2)/100 + cos(x(1))*sin(x(2))/sqrt(2))/sqrt(2);
end
if (nargout == 3)
    H(1,1) = 1/100 + cos(x(1))*cos(x(2))/sqrt(2);
    H(2,1) = -sin(x(1))*sin(x(2))/sqrt(2))/sqrt(2);
    H(1,2) = H(2,1);
    H(2,2) = 1/100 + cos(x(1))*cos(x(2))/sqrt(2))/2;
End
```

Solve this problem from starting points x0. Each column of x0 is a starting point. Call cgobndmin with default options as follows.

```
x0 = 5*rand(2,10);
[xbest, fbest, elapsedtime] = cgobndmin(@griewank, x0, [-10;-10], [10;10])
```

Call cgobndmin with options set as follows.

```
phase1options = optimsetphase1('cgophase1');
phase1options = optimsetphase1('TrialeachTem', 50, 'SAalgorithm', 'saq');
```

```

phase2options = optimset('fmincon');
phase2options = optimset('GradObj', 'on', 'display', 'iter');
otheroptions = optimsetother('cgootheroption');
otheroptions = optimsetother('SaveDir', 'C:\Resultsfolder');
x0 = 5*rand(2,10);
[xbest, fbest, elapsedtime] = cgobndmin(@griewank, x0, [-10;-10], [10;10], phase1options,
phase2options, otheroptions)

```

To use 'sanumavg' in 'SAalgorithm', additional function griewankint.m is needed.

```

function f = griewankint(x1,x2, varargin)

f = (1 + x1.*x1./200 + x2.*x2./200 - cos(x1).*cos(x2./sqrt(2)));

```

To use 'satruavg' in 'SAalgorithm', additional function griewanktrueint.m is needed.

```

function f = griewanktrueint(a,b,c,d, varargin)

f = a*c - a*d - b*c + b*d + (a*c^3)/600 + (a^3*c)/600 - (a*d^3)/600 - (b*c^3)/600 - (a^3*d)/600 -
(b^3*c)/600 + (b*d^3)/600 + (b^3*d)/600 - 2^(1/2)*sin(a)*(sin((2^(1/2)*c)/2) -
sin(1/2*2^(1/2)*d)) + 2^(1/2)*sin(b)*(sin((2^(1/2)*c)/2) - sin(1/2*2^(1/2)*d));

```

Example 2. An illustration of the use of cgobndnls is as follows.

Solve $\min_{-10 \leq x, y \leq 10} \sum_{k=1}^{10} (2 + 2k - e^{kx} - e^{ky})^2$ using cgobndnls with changed options from starting point x0.

Write the objective function in normfun1.m.

```

function [F, J] = normfun1(x,varargin)

k=1:10;
F(k,1) = 2 + 2*k-exp(k*x(1))-exp(k*x(2));
J(k,1) = -k.*exp(k*x(1));
J(k,2) = -k.*exp(k*x(2));

```

Write the approximated function in normfun1approx.m.

```

function F = normfun1approx(x,varargin)

k=1:10;
F(k,1) = 1.9 + 2*k-exp(k*x(1))-exp(k*x(2));

```

Call `cgobndnls` to solve the problem from starting points `x0`.

```
x0 = rand(2, 10);
phase1options = optimsetphase1('cgophase1');
phase1options = optimsetphase1(phase1options, 'UseApproximation', 'on', 'ApproximationFcn',
@normfun1approx);
otheroptions = optimsetother('cgootheroption');
otheroptions = optimsetother(otheroptions, 'UseAdmat', 'on');
```

```
[xbest, fbest, elapsedtime] = cgobndnls(@normfun1, x0, -10*ones(2,1), 10*ones(2, 1),
phase1options, [], otheroptions)
```

Example 3. An illustration of the use of `cgobndquad` is as follows.

Solve $\min_{\substack{-10 \leq x_i \leq 10 \\ -10 \leq x_i \leq 10}} \frac{1}{2} x^T H x + g^T x$ using `cgobndquad` with changed phase one option from thirty

starting points where H is a 6-by-6 matrix and g is a 6-by-1 vector.

```
H = (magic(6)+magic(6)')/2;
g = rand(6, 1);
phase1options = optimsetphase1(phase1options, 'numofeig', 2, 'TrialeachTem', 20);
[xbest, fbest, elapsedtime] = cgobndquad(H, g, -10*ones(6, 1), 10*ones(6, 1), -10+20*rand(6,
10), phase1options)
```