Construction of Global Optimization Constrained NLP Test Cases from Unconstrained Problems

Martin S.C. Chan [a], Ehecatl Antonio del Rio-Chanona [a], Fabio Fiorelli [a], Harvey Arellano-Garcia [b], Vassilios S. Vassiliadis [a*]

[a] *Department of Chemical Engineering and Biotechnology, Pembroke Street, Cambridge CB2 3RA, UK*

[b] *Department of Chemical and Process Engineering, University of Surrey, Guildford, GU2 7XH, UK*

Abstract

This paper presents a novel construction technique for constrained nonconvex Nonlinear Programming Problem (NLP) test cases, derived from the evaluation tree structure of standardized bound constrained problems for which the global solution is known. It is demonstrated in a step-by-step procedure how first an equality constrained problem can be derived from an unconstrained one, with bounds imposed on all variables, using the Directed Acyclic Graph (DAG) of the unconstrained objective function and the use of interval arithmetic to derive bounds for the new variables introduced. An advantage of the proposed methodology is that several standard unconstrained global optimization test cases can be constructed for varying number of optimization variables, thus leading to adjustable size derived NLP's. Further to this in a second step it is demonstrated how any subset of the equalities derived can be relaxed into inequalities giving an equivalent optimization problem. Finally, in a third step it is demonstrated how, by reducing the number of equality constraints derived, it is possible to obtain more complex expressions in the constraints and objective function. The methodology is highlighted throughout by motivating examples and a sample code in *Mathematica*™ is provided in the Appendix.

Keywords: NLP problems, global optimization, constrained nonconvex optimization, unconstrained nonconvex optimization

1. Introduction

Global optimization is of great significance in scientific and engineering practice. The global optimizer may correspond to a structural conformation of a system, such as the folded form of a protein or molecule, which is the lowest energy configuration observed in nature and is the form defining its functional properties. Similarly, in human markets and economic activities, it is obviously important to discover the most economical solution to any routing, design or scheduling problem, as settling down for a local minimum/maximum would represent a loss of utility in comparison to what could be achieved if the global solution was found.

However, global optimization is not an easy task, as even in continuous variable optimization it entails a combinatorial search if the global minimum is to be guaranteed deterministically. Many effective methods for deterministic global optimization have been developed, and references giving extensive coverage of the topic are Floudas [1] and Tawarmalani and Sahinidis [2]. The testing

*Corresponding author. email: vsv20@cam.ac.uk, Fax:(+44)(0)1223334796

of global optimization software, either using deterministic or stochastic methods, requires the availability of test cases of varying complexity and size. For unconstrained global optimization there are numerous standard test functions with known global solutions such as More et al. [3] or Ronkkonen et al. [4], and it is relatively simple to design new ones. However, in the case of constrained global optimization, although some test problems are available [5, 6], it is difficult to procure an assortment of standardized constrained test problems with varying degrees of difficulty.

Our focus in this work is to present a construction technique which transforms any bound constrained problem (bounds may be added without much effort if the global solution is known) into an equivalent constrained Nonlinear Programming Problem (NLP). Equivalence is a term used in the sense of having identical global optimizers to the original problem.

This paper is organized as follows. In section 2 the core of the method proposed is presented. After that, in section 3 an example will be provided, which transforms a well known benchmark unconstrained problem into a constrained equivalent function. Then in section 4 further details are presented to explain how the method was programmed and deployed. Finally, in section 5 the results of the present effort are summarized, underlining the novel contributions. An appendix is provided with the code used in the course of the work.

2. Methodology

The methodology proposed in this work starts with any analytic objective function, corresponding to a bound constrained optimization problem, and analyzes its evaluation tree. We restrict our attention to binary evaluation trees, otherwise known also as Directed Acyclic Graphs (DAG), of the function evaluation. The methodology followed is similar to Smith and Pantelides [7] and Liberti and Pantelides [8], who used DAG's to design a deterministic global optimization method, although our purpose here is to construct constrained NLP test cases of varying complexity and size.

The methodology is based on the fact that any analytic function (*i.e.* which can be expressed as a simple set of instructions on a computer involving arithmetic operations and unary functions – functions involving a single variable as argument) can have its evaluation represented as a binary tree. It is noted that there may be more than one way to represent a function as a binary tree, but they are all equivalent. The binary tree representation requires for each node the evaluation to have two children nodes, a left and a right child, and evaluation proceeds from the leaves of the tree (the actual variables occurring in the function) evaluating higher dependent nodes until the root of the tree is reached (the objective function of the unconstrained problem).

2.1. Deriving NLP's with equality constraints and bounds

To construct the maximal size NLP problem corresponding to the objective function of the unconstrained problem we proceed as follows:

1. Assign a new variable v_i, $i = 1, 2, \ldots, m$ to the m inner nodes of the tree, excluding the objective function evaluation which is at the root of the DAG. We could assign a new variable to the root as well, but that would mean the objective function of the resulting NLP would be comprised simply of that new variable. We propose to avoid having this formulation and rather keep a potentially nonlinear function as the new objective of the resulting NLP.
2. For each new variable in each inner node write down the arithmetic operation it is equal to with evaluation of the node as an equality constraint, for nodes $i = 1, 2, \ldots, m$.
3. Reaching the root of the tree, write down the arithmetic evaluation giving its value as the objective function of the resulting NLP problem.

4. Given bounds for the original variables of the problem, $x_i^L \leq x_i \leq x_i^U$, $i = 1, 2, \ldots, n$ use interval arithmetic to propagate them through the tree to calculate bounds for the variables of the inner nodes, $v_i^L \leq v_i \leq v_i^U$, $i = 1, 2, \ldots, m$. It is noted that bounds for the root of the tree (the objective function) can be derived also in this fashion. It is also noted that bounds derived using interval arithmetic in this manner using the DAG will be loose.
5. Given the known global optimizer of the original optimization problem, x_i^*, $i = 1, 2, \ldots, n$, use the evaluation tree to calculate all the corresponding optimal values of the new variables v_i^*, $i = 1, 2, \ldots, m$ and confirm the objective function value at the optimum.
6. Given the optimal point (x^*, v^*) one may appropriately restrict around it the bounds found in step 4 above.

The procedure thus starts from an original optimization problem given by:

$$\min_{x \in \mathbb{R}^n} f(x) \tag{1a}$$

subject to:

$$x^L \leq x \leq x^U \tag{1b}$$

and derives the following equality constrained NLP problem (with bounds):

$$\min_{x \in \mathbb{R}^n, v \in \mathbb{R}^m} f(x, v) \tag{2a}$$

subject to:

$$h_i(x, v) = 0 \tag{2b}$$

$$x^L \leq x \leq x^U \tag{2c}$$

$$v^L \leq v \leq v^U \tag{2d}$$

The equality constraints $h_i(\cdot, \cdot)$ and the objective function $f(\cdot, \cdot)$ will be of simple form. These will involve either arithmetic operations or power exponentiation between two arguments (the children nodes of the corresponding node i), or a nonlinear function (*e.g.* sin, cos, tan, exp, log, pow) of a single argument. If more complicated forms are desired, it is possible to carry up to node i the expressions of parts of the subtree rooted at node i down to the original variables x of the unconstrained problem. This may be done in such a way that either the number of constraints remains the same, or it is reduced by substituting the entire node expression explicitly for the original variables. The derivation of these more complex constraints is presented in section 3 where a motivating example is used to highlight the methodology, and related options are detailed for its implementation in Mathematica™ which is presented in the section 4.

2.2. Deriving NLP's with equalities, inequalities and bounds

Knowing the global solution of an equality constrained NLP allows one to reformulate some of the equalities into inequalities following analysis of the values of their Lagrange multipliers at the solution.

Compacting the variable vectors into $z = (x^T, v^T)^T$ the Lagrangian of the NLP in equations (2a)–(2d) is given by equation (3), without including the bounds as constraints as it is assumed here that they are inactive for simplicity of presentation:

$$L(z, \lambda) = f(z) + \sum_{i=1}^{m} \lambda_i h_i(z) \tag{3}$$

where λ_i is the Lagrange multiplier associated with equality constraint h_i.

Lagrange multipliers of equality constraints are sign free, while if we had inequality constraints added to the problem such that $h_i(z) \leq 0$ and the corresponding optimal Lagrange multiplier should be $\lambda_i^* \geq 0$. Similarly if we had constraints added to the problem having the form $-h_i(z) \leq 0$ then in the Lagrangian we would have had terms $-\lambda_i h_i(z)$ again with the requirement that $\lambda_i^* \geq 0$.

Thus the equality constrained case is such that if we have a positive multiplier at the solution we can relax the equality constraint as $h_i(z) \leq 0$, or if the multiplier is negative relax it as $-h_i(z) \leq 0$ (or $h_i(z) \geq 0$). In this way we may choose arbitrarily how many of the constraints we wish to write as equalities and how many as inequalities in the resulting NLP.

The procedure for this modification of the resulting NLP is as follows:

1. Set the number of inequalities desired, m_I, such that $m = m_I + m_E$ where m_E is the number of equalities. Choose the partitioning of indices i such that the set I is the set of constraint indices to be treated as inequalities, and E is the set of constraint indices to be treated as equalities.
2. Obtain the optimal Lagrange multipliers for the equality constrained problem, $\lambda^* \in \mathbb{R}^m$, in equations (2a)–(2d).
3. Write the new optimization problem.

The resulting optimization problem is given as:

$$\min_{z \in \mathbb{R}^{n+m}} f(z) \tag{4a}$$

subject to:

$$h_i(z) = 0, \quad \forall\, i \in E \tag{4b}$$

$$\text{sign}\,[\lambda_i^*] \cdot h_i(z) \leq 0, \quad \forall\, i \in I \tag{4c}$$

$$z^L \leq z \leq z^U \tag{4d}$$

where $z^L = \left((x^L)^T, (v^L)^T\right)^T$ and $z^U = \left((x^U)^T, (v^U)^T\right)^T$.

It is noted that the optimal Lagrange multipliers in step 2 above can be obtained either by calling any NLP solver to find a solution for the optimization problem initialized at the known globally optimal solution $z^* = \left((x^*)^T, (v^*)^T\right)$. If the given global solution is trusted then by solving a feasibility problem for the Karush-Kuhn-Tucker (KKT) necessary conditions.

The KKT necessary condition for the problem in equations (2a)–(2d), assuming that the bounds are inactive at the global solution, is given by:

$$\nabla_z f(z^*) + \sum_{i=1}^{m} \lambda_i^* \nabla_z h_i(z^*) = 0 \tag{5}$$

The system in equation (5) , is an overdetermined system of linear equalities. There are m unknowns, the λ^* variables, and the gradient vectors have $n + m$ entries. A suitable pivoting procedure can be used to eliminate the equations so that the m columns have m pivots and the remaining equations are identically equal to zero. The latter condition when there are numerical errors, *e.g.* due to truncation of the reported solution and limited precision of calculating the gradients, will have to be checked within a numerical tolerance in the pivoting (Gaussian elimination) algorithm used.

Alternatively, a feasibility Linear Programming (LP) problem can be set up and solved to find λ^*. For example, using minimization of the infinity norm of the equality violations in the KKT gradient system we get:

$$\min_{\lambda \in \mathbb{R}^m, \varepsilon \in \mathbb{R}^1} \varepsilon \tag{6a}$$

subject to:

$$-\varepsilon \mathbf{1} \leq \nabla_z f(z^*) + \sum_{i=1}^{m} \lambda_i^* \nabla_z h_i(z^*) \leq +\varepsilon \mathbf{1} \tag{6b}$$

$$\varepsilon \geq 0 \tag{6c}$$

where $\mathbf{1} = (1, 1, \ldots, 1)^T \in \mathbb{R}^{n+m}$ is a vector with entries all equal to 1.

The above LP problem has $2(n + m)$ constraints, one bound, and $m + 1$ variables. The value of the objective at the solution, ε^*, is indicative of the satisfaction of the KKT conditions at the given global solution. Determination of the optimal Lagrange multipliers is only necessary to get their correct signs as they define the directionality of the relaxed inequalities of the modified NLP problem.

It is noted that the inequality constraints constructed as above will be active (binding) at the global solution z^*. It is possible to easily include further inactive inequalities (non-binding) at the global solution as follows.

Select a number $m_{I'}$ of arbitrary functions $g_i(z)$ involving any number of the total variables of the system (x and v, indicated by the total vector z), with index i belonging to index set I'. Calculate the values of $g_i(z^*)$ at the globally optimal point and append the following (inactive) inequality constraints to the NLP of equations (4a)–(4d):

$$-\text{sign}\left[g_i(z^*)\right] \cdot g_i(z) \leq 0, \quad \forall\, i \in I' \tag{7}$$

This completes the construction procedure.

3. Motivating example

We consider the Extended Rosenbrock Function (More et al. [3]) for 4 variables:

Table 1: Rosenbrock example equality constraints and optimal Lagrange multipliers

i	$h_i(x)$	λ_i^*	i	$h_i(x)$	λ_i^*
1	$v_1 - x_1^2$	0	10	$v_{10} - x_3^2$	0
2	$v_2 + v_1$	0	11	$v_{11} + v_{10}$	0
3	$v_3 - v_2 - x_2$	0	12	$v_{12} - v_{11} - x_4$	0
4	$v_4 - v_3^2$	-100	13	$v_{13} - v_{12}^2$	-100
5	$v_5 - 100v_4$	-1	14	$v_{14} - 100v_{13}$	-1
6	$v_6 + x_1$	0	15	$v_{15} + x_3$	0
7	$v_7 - v_6 - 1$	0	16	$v_{16} - v_{15} - 1$	0
8	$v_8 - v_7^2$	-1	17	$v_{17} - v_{16}^2$	-1
9	$v_9 - v_5 - v_8$	-1	18	$v_{18} - v_{14} - v_{17}$	-1

$$\begin{aligned} f &= \sum_{i=1}^{2} 100(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2 \\ &= 2 - 2x_1 + x_1^2 + 100x_1^4 - 200x_1^2x_2 + 100x_2^2 \\ &\quad -2x_3 + x_3^2 + 100x_3^4 - 200x_3^2x_4 + 100x_4^2 \end{aligned} \tag{8}$$

A binary tree representation of this function is given in Figure 1 where the interior nodes' variables are indicated to the left of each inner node. The problem results in 27 interior nodes in the binary tree.

The resulting equality constraints are given in Table 1, along with the corresponding optimal Lagrange multipliers. The objective function is $f = v_9 + v_{18}$. The optimal values of all the variables at the global minimum are given in Table 2, along with the intervals calculated for all the variables based on the intervals of the x variables as input.

Finally, additional non-binding constraints at the global solution can be constructed as shown in Table 3. In adding such loose inequalities at the solution, one has to be careful that the variables value range is a subset of the definition domain of the functions introduced.

4. Implementation

The implementation was carried out in *Mathematica*™ (version 8.0), in which operators are overloaded to record the elementary operations (this includes addition, subtraction, division, multiplication and also the exponential, logarithmic and trigonometric functions) when they act on objects with the head `ADOO`. This has been done using the "TagSet" `:/ =` command. When a function is called with an argument of the type `ADOO`, an evaluation trace of the function will be saved to the list "trace", not dissimilar to the trace processed by the reverse accumulation mode of automatic differentiation [9].

Any particular element of the evaluation trace is also a list, corresponding to an elementary function (an intermediate variable). Each will contain the result, index of the arguments of the operation and an arbitrarily assigned operation `ID`. Where an argument of an elementary function is a constant, the value is stored as a string in the trace. The elements are sorted in order of evaluation, with the last element corresponding to the value of the function (and consequently the objective function in the NLP). Essentially, the evaluation trace is a $k \times 4$ array, where k is the number of elementary functions that composes the overall function. The equalities can then be

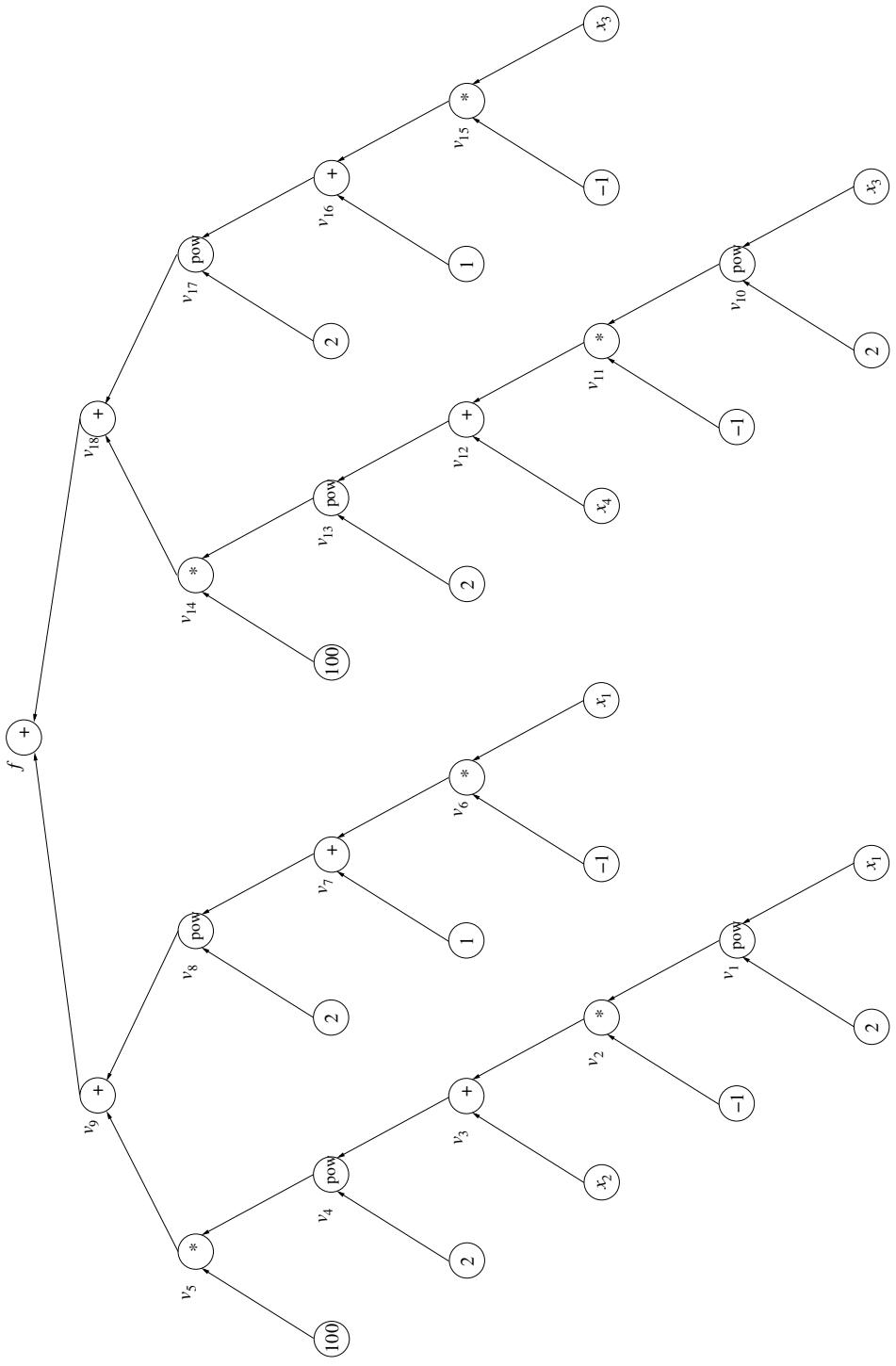

Figure 1: Rosenbrock function binary tree representation

Table 2: Rosenbrock example variable values and intervals

variable	x_1	x_2	x_3	x_4					
optimal value	1	1	1	1					
interval	$[-5, 5]$	$[-5, 5]$	$[-5, 5]$	$[-5, 5]$					
variable	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
optimal value	1	-1	0	0	0	-1	0	0	0
interval	$[0, 25]$	$[-25, 0]$	$[-30, 5]$	$[0, 900]$	$[0, 90000]$	$[-5, 5]$	$[-4, 6]$	$[0, 36]$	$[0, 90036]$
variable	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}	v_{16}	v_{17}	v_{18}
optimal value	1	-1	0	0	0	-1	0	0	0
interval	$[0, 25]$	$[-25, 0]$	$[-30, 5]$	$[0, 900]$	$[0, 90000]$	$[-5, 5]$	$[-4, 6]$	$[0, 36]$	$[0, 90036]$
variable	f								
optimal value	0								
interval	$[0, 180072]$								

Table 3: Rosenbrock example non-binding inequalities construction

i	$g_i(z)$	$g_i(z^*)$	constraint imposed
1	$-\frac{v_2 x_1}{x_3} + e^{-x_3} v_1 x_3$	1.36788	$-g_1(z) \leq 0$
2	$\log[v_{10}]\, v_4 + v_{15} x_1 x_3 x_4$	-1	$+g_2(z) \leq 0$
3	$\sin[x_2 x_4] - \cos[v_5 v_9]\, x_1 x_3$	-0.158529	$+g_3(z) \leq 0$
4	$e^{\frac{v_1}{v_2}}(-x_2 x_4 + x_1 x_2 x_3 x_4) + \tan[v_{11} x_1]$	-1.55741	$+g_4(z) \leq 0$

written as follows: `v[[i]] == trace[[i,1]]` (i.e. the expression of the `i`-th intermediate variable is the `i`-th row of the first column in trace).

There are two optional routines allowing for more variation in the constraints generated. The first is allowing a chosen number of randomly selected constraints to have their expressions collapsed to be in terms of the x variables, whilst deleting the constraints contained in the subtree with the collapsed expression at its root. The routine finds the appropriate constraints to be deleted by the scanning through the information stored in the evaluation trace. Note that this routine will relabel the index of the constraints. Since the constraints are selected simultaneously, it may turn out that some expressions, which were selected to be collapsed, are deleted, resulting in fewer than the chosen number of constants being collapsed. The second option allows the conversion of equalities to inequalities if the optimum solution Lagrange multipliers are provided.

Appendix 5 presents the operator overloading code in *Mathematica*™, which needs to be run so as to render the related subroutines memory-resident. Appendix 5 presents an example of the usage of the code produced. In the example shown, some auxiliary routines are employed to augment the `FindMinimum` optimization solver of *Mathematica*™ which in its current version (version 8.0) does not provide the values of Lagrange multiplier. Although the additional routines are not included in this paper, their operation and output are explained in comments where necessary in the example code. The first printout of the optimization solver has been skipped to avoid excessive cluttering, but the solutions are available later in the execution of the code.

5. Conclusions

This paper presents a methodology for the systematic construction of NLP test problems for global optimization. This is based on analysis of the evaluation tree of standard unconstrained global optimization cases, and construction of appropriate expressions that serve as equality constraints. Using the Lagrange multipliers at the know global optimal solution, it is possible to relax any number of these into binding inequality constraints.

Further options are offered with this methodology to "collapse" any number of nodes in the evaluation tree to produce more complicated algebraic expressions for derived equality constraints. Finally, knowledge of the globally optimal solution allows one to add extra inequality constraints that are by construction designed to be inactive at the solution.

The methodology proposed is an effective way to generate arbitrary size NLP test problems for global optimization studies. This is based on the fact that several standard unconstrained global optimization test cases can be constructed for varying number of optimization variables, thus leading to adjustable size derived NLP's.

Appendix A. *Mathematica*™ code listing

```
(* OL Operators, instead of using b[variable value] to give the variable i.d.
   as before, we instead propagate the i.d. explicitly with the variable. Each
   variable will become a tuple and each operator (except for the 'index'
   operation, the first one) will act on tuples. *)
(* This is the same code as in the reverse mode automatic differentiation
   EXCEPT NOW RECORDS A ELEMENTARY FUNCTION TO TRACE (e.g. v[2]+v[4]) *)
trace = {}; Clear[a]; Clear[c]; Clear[aDOO];
aDOO /: aDOO[x_][[i_]] := {
    c = x[[i]];
    Subscript[v, c] = c;
    aDOO[{c, c}] }[[1]];
aDOO /: aDOO[{x1_, i1_}] + aDOO[{x2_, i2_}] := {
    i++;
    c = Subscript[v, i1] + Subscript[v, i2];
    trace = Append[trace, {c, i1, i2, 1}];
    aDOO[{c, i}] }[[1]];
aDOO /: aDOO[{x1_, i1_}] - aDOO[{x2_, i2_}] := {
    i++;
    c = Subscript[v, i1] - Subscript[v, i2];
    trace = Append[trace, {c, i1, i2, 2}];
    aDOO[{c, i}] }[[1]];
aDOO /: aDOO[{x1_, i1_}]*aDOO[{x2_, i2_}] := {
    i++;
    c = Subscript[v, i1]*Subscript[v, i2];
    trace = Append[trace, {c, i1, i2, 3}];
    aDOO[{c, i}] }[[1]];
aDOO /: aDOO[{x1_, i1_}]^aDOO[{x2_, i2_}] := {
    i++;
    c = Subscript[v, i1]^Subscript[v, i2];
    trace = Append[trace, {c, i1, i2, 4}];
    aDOO[{c, i}] }[[1]];
aDOO /: Sin[aDOO[{x1_, i1_}]] := {
    i++;
    c = Sin[Subscript[v, i1]];
    trace = Append[trace, {c, i1, Null, 5}];
    aDOO[{c, i}] }[[1]];
aDOO /: Cos[aDOO[{x1_, i1_}]] := {
    i++;
    c = Cos[Subscript[v, i1]];
    trace = Append[trace, {c, i1, Null, 6}];
    aDOO[{c, i}] }[[1]];
aDOO /: Exp[aDOO[{x1_, i1_}]] := {
    i++;
    c = Exp[Subscript[v, i1]];
    trace = Append[trace, {c, i1, Null, 7}];
    aDOO[{c, i}] }[[1]];
aDOO /: Log[aDOO[{x1_, i1_}]] := {
    i++;
    c = Log[Subscript[v, i1]];
    trace = Append[trace, {c, i1, Null, 9}];
    aDOO[{c, i}] }[[1]];
aDOO /: aDOO[{x1_, i1_}] + x2_ := {
    i++;
    c = Subscript[v, i1] + x2;
    trace = Append[trace, {c, i1, ToString[x2] <> "@", 1}];
```

```
    aDOO[{c, i}] }[[1]];
aDOO /: aDOO[{x1_, i1_}] - x2_ := {
    i++;
    c = Subscript[v, i1] - x2;
    trace = Append[trace, {c, i1, ToString[x2] <> "@", 2}];
    aDOO[{c, i}] }[[1]];
aDOO /: aDOO[{x1_, i1_}]*x2_ := {
    i++;
    c = Subscript[v, i1]*x2;
    trace = Append[trace, {c, i1, ToString[x2] <> "@", 3}];
    aDOO[{c, i}] }[[1]];
aDOO /: aDOO[{x1_, i1_}]^n_ := {
    i++;
    c = Subscript[v, i1]^n;
    trace = Append[trace, {c, i1, ToString[n] <> "@", 4}];
    aDOO[{c, i}] }[[1]];
aDOO /: Value[aDOO[x_]] := x;
aDOO /: Dimensions[aDOO[x_]] := Dimensions[x];

(* This routine uses overloaded operators to load function and input variables
   in order to produce the evaluation trace for subsequent routines to analyse
   *)
  loadTrace[function_, vars_] := Block[{trace = {}, a, i = 0, c, CPUTime},

  (* sample the CPU time *)
  CPUTime = TimeUsed[];

  function[aDOO[vars]];

  (* sample the CPU time from start of run *)
  CPUTime = TimeUsed[] - CPUTime;

  Print["=================================================="];
  Print["Function " <> ToString[function] <> " loaded\n"];
  Print["Number of operations   = ", Length[trace]];
  Print["\nCPU time (seconds)     = ", CPUTime];
  Print["=================================================="];

  trace ]

  (* This routine analyses "trace" and outputs the first pass constraints in
     the form h_i(v)=0 *)
  generateConstraints1[trace_] :=
    Block[{expressions = {}, equations, constraints, CPUTime, i, objective},

  (* sample the CPU time *)
  CPUTime = TimeUsed[];

  (* Write each constraint in the form "Subscript[h, i](v)==0" into "equations"
      *)
  For[i = 1, i < Length[trace], i++,
     AppendTo[expressions, Subscript[v, i] - trace[[i, 1]]] ];

  equations = Thread[(expressions) == 0];

  (* Label each constraint with "Subscript[h, i]" *)
```

```
  constraints = Transpose[{Table["h" <> ToString[i], {i, Length[equations]}],
     equations }];

  (* Extract the objective function *)
  objective = trace[[Length[trace], 1]];

  (* sample the CPU time from start of run *)
  CPUTime = TimeUsed[] - CPUTime;

  Print["=================================================="];
  Print["First pass constraint statistics\n"];
  Print["Objective function    = ", objective];
  Print["Number of variables   = ", Length[constraints],
   " intermediate ", "+ ", Length[xvars], " input"];
  Print["Number of constraints = ", Length[constraints]];
  Print["\nCPU time (seconds)    = ", CPUTime];
  Print["=================================================="];

  {constraints, objective} ]

(* This routine replaces Mathematica's in-built Solve[] function to produce the
     list "solvsubst". This seems to be substantially faster than Solve[]. I
    felt that "solvsubst" should be global variable as it is called for in a few
     separate routines. This requires "trace" to be loaded. *)
  varSolve[trace_] := Block[{vvars, i, n, temp, solvsubst, CPUTime},

  (* sample the CPU time *)
  CPUTime = TimeUsed[];

  (* Number of variables "n" *)
  n = Length[trace] - 1;

  (* Create temporary list of variables "temp" which will contain the collapsed
      expressions.
  This essentially replaces the Solve[] function, and is a lot faster *)
  vvars = Table[Subscript[v, i], {i, n}];

  For[i = 1, i <= n, i++,
      Subscript[v, i] = trace[[i, 1]] ];

  temp = vvars; Clear[Subscript];

  (* Create substitution list "solvsubst", "Subscript[v, i] -> (x variables)"
     *)

  solvsubst = {};

  For[i = 1, i <= n, i++,
     solvsubst = Append[solvsubst, Subscript[v, i] -> temp[[i]]] ];

  (* sample the CPU time from start of run *)
  CPUTime = TimeUsed[] - CPUTime;

  Print["=================================================="];
  Print["Substitution rules for intermediate variables in terms of input
     variables\n"];
```

```
  Print["Number of input variables   = ", Length[xvars]];
  Print["Number of intermediate variables   = ", Length[vvars]];
  Print["\nCPU time (seconds)    = ", CPUTime];
  Print["====================================================="];

  solvsubst ]

(* This routine finds the optimal solution point for the intermediate variables
    given the known solution point of the x variables. This also requires "
   trace" and "solvsubst" to be loaded. *)
optimalSolutionPoint[trace_, xvars_, xvals_, solvsubst_] :=
  Block[{i, Subscript, vvars, n, substintervvalx, expressions, equations,
     OptimalValsTable, OptimalIntervTable, CPUTime, temp, substx},

  (* sample the CPU time *)
  CPUTime = TimeUsed[];

  (* Number of variables "n" *)
  n = Length[trace] - 1;

  (* Create temporary list of variables "temp" which will contain the collapsed
      expressions.
  This essentially replaces the Solve[] function, and is a lot faster *)
  vvars = Table[Subscript[v, i], {i, n}];

  (* Calculate and write optimal variable values to "OptimalValsTable",
     alongside their respective variable symbols *)
  substx = Thread[xvars -> xvals];
  OptimalValsTable = Transpose[{Join[xvars, vvars], Join[xvars, vvars] /.
     solvsubst /. substx }];

  (* sample the CPU time from start of run *)
  CPUTime = TimeUsed[] - CPUTime;

  Print["====================================================="];
  Print["Optimal solution point\n"];
  Print["Number of variables   = ", Length[OptimalValsTable]];
  Print["\nCPU time (seconds)    = ", CPUTime];
  Print["====================================================="];

  OptimalValsTable ]

(* This routine finds the intervals of the intermediate variables given the
    intervals of the x variables. This also requires "trace" and "solvsubst" to
    be loaded. *)
variableIntervals[trace_, xvars_, substintervx_, solvsubst_] :=
   Block[{i, Subscript, vvars, n, substintervvalx, intervalallsubst,
      expressions, equations, OptimalValsTable, OptimalIntervTable, CPUTime,
      temp},

  (* sample the CPU time *)
  CPUTime = TimeUsed[];

  (* Number of variables "n" *)
  n = Length[trace] - 1;
```

```
  (* Create temporary list of variables "temp" which will contain the collapsed
      expressions. This essentially replaces the Solve[] function, and is a lot
      faster *)
  vvars = Table[Subscript[v, i], {i, n}];

  (* Find intervals of all variables given intervals of the initial variables
     *)
  substintervvalx = Thread[xvars -> substintervx];

  intervalallsubst = Join[substintervvalx, solvsubst /. substintervvalx];

  OptimalIntervTable = Transpose[{Join[xvars, vvars], Join[xvars, vvars] /.
     intervalallsubst}];

  (* sample the CPU time from start of run *)
  CPUTime = TimeUsed[] - CPUTime;

  Print["=================================================="];
  Print["Variable intervals\n"];
  Print["Number of intervals = ", Length[OptimalIntervTable]];
  Print["\nCPU time (seconds)    = ", CPUTime];
  Print["=================================================="];

  OptimalIntervTable]

(* This routine will collapse variables (expressing them in terms of the x
    variables). The variables will be selected randomly, the user specifies the
    number of variables in the "number" argument when calling the routine. The
    resulting redundant variables will then be identified and deleted. This also
     requires "trace" and "solvsubst" to be loaded. Note that in some cases,
    variables that are selected to be collapsed may also be deleted as a result
    of other variables being selected (this becomes more probable as the "number
    " becomes larger). *)

collapseConstraints[trace_, constraints_, solvsubst_, number_] :=
  Block[{variationIDs, k, i, deletevars = {}, tempconstraints, CPUTime, scan,
     dep},

   (* sample the CPU time *)
   CPUTime = TimeUsed[];

   variationIDs = RandomSample[Table[i, {i, Length[constraints]}], number];

   (* This loop performs the substitution for each variable ID stored in the
      list "variationIDs" *)
   tempconstraints = constraints;

   For[i = 1, i <= Length[variationIDs], i++,
      k = variationIDs[[i]];
      tempconstraints[[k, 2]] = Subscript[v, k] - (Subscript[v, k] /. solvsubst
         [[k]]) == 0 ];

  (* ==================================== *)
  (* Now define this function that will produce the intermediate variables that
      a given variable is dependent on "scan[j]" looks at the j'th row in the
     array "trace", then at the 2nd and 3rd columns in that row. If those
```

```
      elements are integers, that means that that Subscript[v, j] is dependent
      on those elements, and records the elements to "list". It then repeats
      this operation for each of those elements. *)
  scan[j_] := Block[{f},
    f = trace[[j, 2]];
    If[IntegerQ[f], {AppendTo[list, f], scan[f]}];

    f = trace[[j, 3]];
    If[IntegerQ[f], {AppendTo[list, f], scan[f]}]; ];

  (* "dep[i]" produces the list of variables that Subscript[v, i] depends on *)
  dep[i_] := Block[{list = {}}, scan[i]; list ];

  (* ===================================== *)
  (* Using the function that has just been defined above, this loop records
     which variables should be deleted, as they have been 'carried up' to
     variables that have been 'lumped' *)
  For[i = 1, i <= Length[variationIDs], i++,
     k = variationIDs[[i]];
     deletevars = Join[deletevars, dep[k]];
     deletevars = Union[deletevars] ];

  (* Deletes the redundant variables identified by "deletevars" *)
  tempconstraints = Delete[tempconstraints, Thread[{deletevars}]];

  (* Relabel the Subscript[h, i] labels in constraints from 1 to the new number
      of constraints *)
  tempconstraints = Delete[Transpose[tempconstraints], {1}];

  tempconstraints = Transpose[{Table["h" <> ToString[i],
    {i, Length[tempconstraints[[1]]]}], tempconstraints[[1]]}];

  (* sample the CPU time from start of run *)
  CPUTime = TimeUsed[] - CPUTime;

  Print["====================================================="];
  Print["Collapsing variables (and subsequently deleting variables)
    \n"];
  Print["Number of variables collapsed = ", number];
  Print["Former IDs of variables collapsed = ", Sort[variationIDs]];
  Print["Number of variables deleted = ", Length[deletevars]];
  Print["Former IDs of variables deleted = ", Sort[deletevars]];
  Print["New number of constraints = ", Length[tempconstraints]];
  Print["\nCPU time (seconds)    = ", CPUTime];
  Print["====================================================="];

  {tempconstraints, deletevars} ]

(* This section choose "number" of constraints to be converted to inequalities,
    given their respective Lagrange multipliers. This also requires "trace", "
   constraints", "lagrangemultipliers" to be loaded. "deletevars" is only
   required if constraints have been deleted from the collapse expressions
   routine, input 0 if this is not the case. *)

inequalityConstraints[trace_, constraints_, lagrangemultipliers_, number_,
   deletevars_] :=
```

```
Block[{equationIDs, inequalityIDs, k, temp, expressions = {},
 deletedexpressions, CPUTime, zeromultipliers = {}},

 (* sample the CPU time *)
 CPUTime = TimeUsed[];

 (* Write each expression to "expressions" from "trace" *)
 For[i = 1, i < Length[trace], i++,
    AppendTo[expressions, Subscript[v, i] - trace[[i, 1]]] ];

 (* This decides if deletevars is a list or 0, and sets deletedexpression to
     the appropriate case *)
 If[ListQ[deletevars],
    deletedexpressions = Delete[expressions, Thread[{deletevars}]],
    deletedexpressions = expressions ];

 (* Choose randomly the i'th constraints that will be converted to
    inequalities *)
 equationIDs = Table[i, {i, Length[constraints]}];
 inequalityIDs = RandomSample[equationIDs, number];

 (* This alters the i'th entry of the list "temp" (which is actually "
    constraints") to become inequalities, as chosen by the list "
    inequalityIDs" *)
 temp = constraints;
 For[i = 1, i <= Length[inequalityIDs], i++, k = inequalityIDs[[i]];

 (* If the lagrange multiplier is zero, do not change the respective
    constraint, leave it as an equality and save the ID to "zeromultipliers"
     *)
  If[lagrangemultipliers[[k]] == 0,
     AppendTo[zeromultipliers, k],
     temp[[k, 2]] = Sign[lagrangemultipliers[[k]]]*deletedexpressions[[k]]
        <= 0 ] ];

 (* sample the CPU time from start of run *)
  CPUTime = TimeUsed[] - CPUTime;

 Print["=================================================="];
 Print["Converting equalities to inequalities\n"];
 Print["Number of inequalities converted = ",
    number - Length[zeromultipliers]];
 Print["IDs of equalities converted to inequalities = ",
    Sort[inequalityIDs]];
 Print["IDs of equalities that have a Lagrange multiplier
    of zero = ", Sort[zeromultipliers]];
 Print["Number of constraints = ", Length[temp]];
 Print["\nCPU time (seconds)    = ", CPUTime];
 Print["=================================================="];

 temp]
```

Appendix B. Example run

```
     * Initiate input variables and define function to be analysed in this
        Block. The contents of this Block is specific only to our example
        problem *)
```

```
Block[{n},
(* Choose number of variables "n" *) n=24;
(       * Define x variables, their optimum values and their bounds *)
   xvals=Table[1,{n}]; xvars=Table[Subscript[x, i],{i,1,n}];
   substintervx=Table[Interval[{-5,5}],{Length[xvars]}];
(* This is the Rosenbrock function with "n" variables *)
RF[n_,x_]:=\!\( \*UnderoverscriptBox[\(\[Sum]\), \(j = 1\), \(n/2\)
   ]\((100*\((x[\([\)\(2  j\)\(]\)] - x[\([\)\(2  j - 1\)\(]\)]^2)\)^2
   + \((1 - x[\([\)\(\((2  j)\) - 1\)\(]\)])\)^2)\)\);
f[x_]:=Expand[Simplify[RF[Length[xvars],x]]]; ]

(* Execute routine and save into "trace" *)
trace=loadTrace[f,xvars];

=================================================================
Function f loaded
Number of operations   = 119
CPU time (seconds)     = 0.015
=================================================================

(*" Execute routine and save constraints into "constraints", and the
   objective function into "objective" "*)    {constraints,objective}=
   generateConstraints1[trace];

=================================================================
First pass constraint statistics
Objective function    = Subscript[v, 116]+Subscript[v, 118]
Number of variables   = 118 intermediate + 24 input
Number of constraints = 118
CPU time (seconds)    = 0.
=================================================================

(*" Execute routine and save the substitution rules into "solvsubst",
   and the objective function into "objective" "*)   solvsubst=varSolve
   [trace];

"================================================================="
"Substitution rules for intermediate variables in terms of input
   variables\n"
"Number of input variables   = "24
"Number of intermediate variables   = "118
"\nCPU time (seconds)    = "0.04699999999999993‘
"================================================================="

(* Execute routine and save optimal variable values into "optimalvalues
   " *)     optimalvalues=optimalSolutionPoint[trace,xvars,xvals,
   solvsubst];
   "================================================================="
"Optimal solution point\n" "Number of variables   = "142
"\nCPU time (seconds)    = "0.016000000000000014‘
"================================================================="

(* This creates the "variables" list in the right format to be used by
   the NLP solver *)
Thread[Insert[Transpose[optimalvalues],-INF,2]];
variables=Thread[Insert[Transpose[%],INF,3]];
```

```
(* Execute routine and save variable intervals into "intervals" *)
      intervals=variableIntervals[trace,xvars,substintervx,solvsubst
   ];
   "================================================================"
"Variable intervals\n" "Number of intervals = "142
"\nCPU time (seconds)    = "0.‘
"================================================================"

(* Execute routine and save new set of constraints into "newconstraints
   ". As an example, the number of variables collapsed is set to 3 here
   . A list of the deleted variables is saved for use in the inequality
    section. *)     {newconstraintscollapsed,deletevars}=
   collapseConstraints[trace,constraints,solvsubst,3];
(* Uncomment the line below to analyse "newconstraints" in the NLP
   solver (or alternatively change "constraints" to "
   newconstraintscollapsed" in the argument of NLPParse[] *)
constraints=newconstraintscollapsed;
"================================================================"
"Collapsing variables (and subsequently deleting variables)\n"
"Number of variables collapsed = "3
"Former IDs of variables collapsed = "{82,83,118}
"Number of variables deleted = "89
"Former IDs of variables deleted =      "
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,

49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,

73,74,75,76,77,78,79,80,81,82,109,110,111,112,113,114,117}
"New number of constraints = "29
"\nCPU time (seconds)    = "0.015000000000000124‘
"================================================================"

(* The following set of cells is simply using the NLP solver to get the
    Lagrange multipliers. The required objects "objective", "
   constraints", "variables" have already been created prior to this
   cell. The rest of the required objects are created here now. *)
paramvalues={};
parameters={};

(* Call the parser to parse the model *) {ProblemModel,ProblemVariables
   }=NLPParse["Test1",objective,constraints,variables,parameters,True];

   "================================================================"
"NLP PARSER INTERFACE FOR FindMinimum\n" "NLP Problem Name = "
"Test1"
"\n" "Number of parameters  = "0
"Number of variables   = "142
"Number of constraints = "29
"Number of bounds      = "0
"\nCPU time (seconds)    = "0.016000000000000014‘
"================================================================"

(* Find solution of the optimization problem *)
{ProblemObjOptValue, VarSubstitutionOptSol}= FindMinimum[
   ProblemModel@@paramvalues,ProblemVariables@@paramvalues,AccuracyGoal
```

```
   ->{8,10},PrecisionGoal->10,MaxIterations->2000 ]

(* Extract solution into a vector *)
sol=NLPSubstListToVector[VarSubstitutionOptSol]
{1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,
1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,1.‘,-1.‘,0.‘,
0.‘,0.‘,-1.‘,0.‘,0.‘,0.‘,1.‘,-1.‘,0.‘,0.‘,0.‘,-1.‘,0.‘,
0.‘,0.‘,1.‘,-1.‘,0.‘,0.‘,0.‘,-1.‘,0.‘,0.‘,0.‘,1.‘,-1.‘,
0.‘,0.‘,0.‘,-1.‘,0.‘,0.‘,0.‘,1.‘,-1.‘,0.‘,0.‘,0.‘,-1.‘,
0.‘,0.‘,0.‘,1.‘,-1.‘,0.‘,0.‘,0.‘,-1.‘,0.‘,0.‘,0.‘,1.‘,
-1.‘,0.‘,0.‘,0.‘,-1.‘,0.‘,0.‘,0.‘,1.‘,-1.‘,0.‘,0.‘,0.‘,
-1.‘,0.‘,0.‘,0.‘,1.‘,-1.‘,0.‘,0.‘,0.‘,-1.‘,0.‘,0.‘,0.‘,
1.‘,-1.‘,0.‘,-6.902532920683853‘*^-32,-6.902532920683853‘*^-30,
-1.‘,0.‘,0.‘,-6.902532920683853‘*^-30,1.‘,-1.‘,0.‘,
9.860761315262648‘*^-33,9.860761315262648‘*^-31,-1.‘,0.‘,0.‘,
9.860761315262648‘*^-31, 1.‘,-1.‘,0.‘,9.860761315262648‘*^-33,
9.860761315262648‘*^-31,-1.‘,0.‘,0.‘,9.860761315262648‘*^-31,0.‘
,0.‘,0.‘,0.‘,0.‘,0.‘,-5.9164567891575885‘*^-30,
-4.930380657631324‘*^-30,0.‘,0.‘}

NLPObjectiveConstraintParser["Test1",True];

"================================================================="
"NLP CONSTRAINTS AND OBJECTIVE PARSER\n"
"NLP Problem Name = ""Test1""\n"
"Number of constraints = "29
"       of which \"==\" are = "29
"       of which \"<=\" are = "0 "
of which \">=\" are = "0
"Number of bounds      = "0
"       of which lower bounds are = "0
"       of which upper bounds are = "0
"\nCPU time (seconds)    = "0.016000000000000014‘
"================================================================="

(* Call the Constraint and objective gradient setup routine *)
NLPGradientParser["Test1",True];
"================================================================="
"NLP CONSTRAINTS AND OBJECTIVE GRADIENTS PARSER\n"
"NLP Problem Name = "
"Test1""\n"
"Number of variables   = "142
"Number of constraints = "29
"Number objective   gradient nonzeros = "2
"Number constraints Jacobian nonzeros = "83
"\nCPU time (seconds)    = "0.‘
"================================================================="
(* display things *) Print[NLPModel["Test1"]["
   ConstraintGradientVarIncidence"]]; Print[NLPModel["Test1"]["
   ObjectiveGradientVarIncidence"]];
{{1,19},{1,107},{2,20},{2,107},{2,108},{3,108},{3,109},{4,109},
{4,110},{5,19},{5,111},{6,111},{6,112},{7,112},{7,113},{8,110},
{8,113},{8,114},{9,21},{9,115},{10,115},{10,116},{11,22},{11,116},
{11,117},{12,117},{12,118},{13,118},{13,119},{14,21},{14,120},
{15,120},{15,121},{16,121},{16,122},{17,119},{17,122},{17,123},
{18,23},{18,124},{19,124},{19,125},{20,24},{20,125},{20,126},
```

```
{21,126},{21,127},{22,127},{22,128},{23,23},{23,129},{24,129},
{24,130},{25,130},{25,131},{26,128},{26,131},{26,132},{27,114},
{27,123},{27,139},{28,132},{28,139},{28,140},{29,1},{29,2},
{29,3},{29,4},{29,5},{29,6},{29,7},{29,8},{29,9},{29,10},{29,11},
{29,12},{29,13},{29,14},{29,15},{29,16},{29,17},{29,18},{29,142}}
{140,142}
NLPModel["Test1"]["ConstGradNumFast"]@@Join[sol,paramvalues]
{2.`,1.`,-1.`,-1.`,1.`,0.`,1.`,-100.`,1.`,1.`,1.`,-1.`,1.`,0.`
,1.`,-1.`,-1.`,1.`,-2.`,1.`,1.`,1.`,-1.`,-1.`,1.`,0.`,1.`,-100.`,
1.`,1.`,1.`,-1.`,1.`,0.`,1.`,-1.`,-1.`,1.`,-2.`,1.`,1.`,1.`,-1.`,
-1.`,1.`,0.`,1.`,-100.`,1.`,1.`,1.`,-1.`,1.`,0.`,1.`,-1.`,-1.`,1.`,
-1.`,-1.`,1.`,-1.`,-1.`,1.`,0.`,0.`,0.`,0.`,0.`,0.`,0.`,0.`,0.`,0.`,
0.`,0.`,0.`,0.`,0.`,0.`,0.`,0.`,1.`}

NLPModel["Test1"]["ObjGradNumFast"]@@Join[sol,paramvalues] {1,1}
(* Call the solution analyser to find out the active constraints
(without Lagrange multiplier calculation) *)

NLPModelEvaluation["Test1",sol,paramvalues,N[10^-6],True]
"======================================================"
"CONSTRAINT ANALYSIS ROUTINE FOR GIVEN\nVARIABLES AND PARAMETERS VALUES
   \n"
"Number of active constraints = "29
"Number of total  constraints = "29
"Constraint norm              = "8.874685183736382`*^-32
"\n"
"Number of active bounds constraints = "0
"Number of total  bounds constraints = "0
"Bounds norm                         = "0.`
"\nCPU time (seconds)    = "0.015000000000000124`
"======================================================"

(* display some things *)
Print[NLPModel["Test1"]["BoundsActivity"]];
Print[NLPModel["Test1"]["ConstraintActivity"]];
{}
{{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
17,18,19,20,21,22,23,24,25,26,27,28,29},{},{}}
NLPModelLagrangeMultipliers["Test1",True]
"======================================================"
"LAGRANGE MULTIPLIER CALCULATION ROUTINE\n"
"Successful completion."
"    Fitting tolerance returned = "2.2440382885235977`*^-13
"    Number of constraint multipliers = "29
"    Number of bounds     multipliers = "0
"\nCPU time (seconds)    = "0.016000000000000014`
"======================================================" True
(* display some things *)
Print[NLPModel["Test1"]["ConstraintLagrangeMultipliers"]];
Print[NLPModel["Test1"]["BoundsLagrangeMultipliers"]];
{-1.1220191442617988`*^-13,1.26329502414535`*^-13,
-99.99999999988583`,-0.9999999999988903`,
4.4880765770471953`*^-13,2.2440382885235977`*^-13,
-0.9999999999988965`,-0.9999999999991618`,
-1.1220191442617988`*^-13,-1.3063161663495748`*^-13,
-2.2440382885235977`*^-13,-99.9999999998905`,
```

```
-0.9999999999989071`,-4.4880765770471953`*^-13,
-2.2440382885235977`*^-13,-0.9999999999988965`,
-0.9999999999991597`,-1.1220191442617988`*^-13,
-1.1933509735939651`*^-13,-2.2440382885235977`*^-13,
-99.99999999992002`,-0.9999999999992026`,
-4.4880765770471953`*^-13,-2.2440382885235977`*^-13,
-0.9999999999997154`,-0.9999999999994227`,
-0.9999999999994307`,-0.9999999999996991`,
-0.9999999999997756`} {}
Transpose[{
        Table[i,{i,Length[constraints]}],
        NLPModel["Test1"]["ConstraintLagrangeMultipliers"] } ]
{{1,-1.1220191442617988`*^-13},{2,1.26329502414535`*^-13},
{3,-99.99999999988583`},{4,-0.9999999999988903`},
{5,4.4880765770471953`*^-13},{6,2.2440382885235977`*^-13},
{7,-0.9999999999988965`},{8,-0.9999999999991618`},
{9,-1.1220191442617988`*^-13},{10,-1.3063161663495748`*^-13},
{11,-2.2440382885235977`*^-13},{12,-99.9999999998905`},
{13,-0.9999999999989071`},{14,-4.4880765770471953`*^-13},
{15,-2.2440382885235977`*^-13},{16,-0.9999999999988965`},
{17,-0.9999999999991597`},{18,-1.1220191442617988`*^-13},
{19,-1.1933509735939651`*^-13},{20,-2.2440382885235977`*^-13},
{21,-99.99999999992002`},{22,-0.9999999999992026`},
{23,-4.4880765770471953`*^-13},{24,-2.2440382885235977`*^-13},
{25,-0.9999999999997154`},{26,-0.9999999999994227`},
{27,-0.9999999999994307`},{28,-0.9999999999996991`},
{29,-0.9999999999997756`}}

(* Save Lagrange multipliers to a list to be accessed by the inequality
    generator *) lagrangemultipliers=NLPModel["Test1"]["
   ConstraintLagrangeMultipliers"];
(* Executes routine and saves the updated set of constraints to "
   newconstraintsinequalities".
Note some constraints became "True" if their respective Lagrange
   multiplier = 0, these entries have been deleted. *)
   newconstraintsinequalities=inequalityConstraints[trace,constraints,
   lagrangemultipliers,3,deletevars];

(* Uncomment the line below to analyse "newconstraintsinequalities" in
   the NLP solver (or alternatively change "constraints" to "
   newconstraintsinequalities" in the argument of NLPParse[] *)
constraints=newconstraintsinequalities;
   "================================================================"
"Converting equalities to inequalities\n"
"Number of inequalities converted = "3
"IDs of equalities converted to inequalities = "{3,26,29}
"IDs of equalities that have a Lagrange multiplier of zero = "{}
"Number of constraints = "29
"\nCPU time (seconds)    = "0.`
"================================================================"
```

References

[1] C. Floudas, Deterministic Global Optimization: Theory, Methods and Applications, Kluwer Academic Publishers, 2000.

[2] M. Tawarmalani, N. Sahinidis, Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming, Kluwer Academic Publishers, 2002.

[3] J. More, B. Garbow, K. Hillstrom, Testing Unconstrained Optimization Software, ACM Transactions on Mathematical Software 7 (1981) 17–41.

[4] J. Ronkkonen, X. Li, V. Kyrki, J. Lampinen, A Framework for Generating Tunable Test Functions for Multimodal Optimization, Soft Computing - A Fusion of Foundations, Methodologies and Applications - Special Issue on Evolutionary Optimization and Learning 15 (2011) 1689–1706.

[5] The Optimization Firm, NLP and MINLP test problems, URL `http://www.minlp.com/nlp-and-minlp-test-problems`, 2016.

[6] A. Neumaier, Global Optimization Test Problems, URL `http://www.mat.univie.ac.at/ neum/glopt/test.html`, 2016.

[7] E. Smith, C. Pantelides, A Symbolic Reformulation/Spatial Branch-and-Bound Algorithm for the Global Optimisation of Nonconvex MINLPs, Computers and Chemical Engineering 23 (1999) 457–478.

[8] L. Liberti, C. Pantelides, An Exact Reformulation Algorithm for Large Nonconvex NLPs Involving Bilinear Terms, Journal of Global Optimization 36 (2006) 1689–1706.

[9] G. Corliss, A. Griewank, Operator Overloading as an Enabling Technology for Automatic Differentiation, Tech. Rep. CRPC-TR93431, Center for Research for Parallel Computing, Rice University, Houston,TX,USA, 1993.