User's Guide for FORTRAN Global Optimisation Code NLPGLOB

B. Chachuat, M.A. Latifi

June 2002

Abstract

NLPGLOB is a set of FORTRAN subroutines for the global optimisation of general twice-differentiable constrained NLP problems.

NLPGLOB implements a branch-and-bound algorithm. Upper bounds are obtained through repeated solution to local optimisation problems. On the other hand, lower bounds rely on the notion of a convex underestimator for a nonconvex function. Because convex underestimating problems can be solved to guaranteed global optimality, they are employed to generate rigorous lower bounds on nonconvex problems. For this reason, convex underestimators are ideally suited as the relaxations required by a deterministic global branch-and-bound algorithm for solving nonconvex NLPs. Tight underestimators for nonconvex terms of special structure (such as bilinear, fractional, trilinear, trilinear fractional and univariate concave terms) are used to enhance the convergence of the algorithm. Convex underestimators for any other general twice-differentiable nonconvex terms are constructed via a shift of the diagonal element of the Hessian matrix of the nonconvex function, and values for the shift parameter α are computed via interval analysis tools.

To use NLPGLOB the user is required to provide subroutines that define (*i*) the objective and constraint functions of both nonconvex and convex underestimated problems and, (*ii*) the gradients of these functions (unless NLPGLOB is required to estimate these gradients by finite differences). Note that further information is needed by the code to deal with nonconvex terms of special structure as well as general twice differentiable nonconvex terms. Therefore, NLPGLOB is suited for small nonconvex problems (*e.g.* up to 10 parameters and/or constraints). Enhancements of this package are currently under consideration that consist in implementing automatic differentiation and interval computation tools.

Contact

Prof. M.A. Latifi Laboratoire des Sciences du Génie Chimique, CNRS–ENSIC, B.P. 451, 1 rue Grandville, 54001 Nancy Cedex.
email: `latifi@ensic.inpl-nancy.fr`, tel: +33 (0) 383 175 234, fax: +33 (0) 383 175 326

Dr. B. Chachuat Laboratoire des Sciences du Génie Chimique, CNRS–ENSIC, B.P. 451, 1 rue Grandville, 54001 Nancy Cedex.
email: `latifi@ensic.inpl-nancy.fr`, tel: +33 (0) 383 175 180, fax: +33 (0) 383 175 326

CONTENTS

Introduction

Whenever a mathematical model is available to simulate a real-life application, a straightforward idea is to apply mathematical optimisation algorithms for minimising a so-called cost function subject to constraints. Highly complex industrial and academic design problems are solved today by means of nonlinear programming algorithms without any chance to get equally qualified results by traditional empirical approaches. However, it is well known for a number of years that many nonlinear problems encountered in chemical engineering applications exhibit multiple local optima. This property which can be attributed to nonconvexity of the functions participating in most chemical engineering models, implies that standard local optimisation methods will often yield suboptimal solutions to problems and has motivated researchers to develop global optimisation algorithms for solving nonconvex NLPs.

Of particular interest are deterministic global optimisation algorithms, or these methods rigorously proving the finite convergence of the algorithm to global optimality. The usage of these algorithms therefore guarantees that the optimal parameter values have been found for the considered formulation. Many modern, general methods for deterministic global optimisation in Euclidean spaces rely on the notion of a convex underestimator for a nonconvex function [McCormick, 1976]. As the name implies, a convex underestimator is a convex function that underestimates a nonconvex function on the set of interest. Because convex underestimated problems can be solved to guaranteed global optimality, they are employed to generate rigorous lower bounds on nonconvex problems. For this reason, convex underestimators are ideally suited as the relaxations required by a deterministic global branch-and-bound algorithm for solving nonconvex NLPs [McCormick, 1976, Horst and Tuy, 1996].

Floudas and co-workers have demonstrated that a convex underestimator can be constructed for any twice-differentiable nonconvex function via a shift of the diagonal elements of the Hessian matrix of the nonconvex function [Maranas and Floudas, 1994]. The shift parameter, denoted α, is utilised in conjunction with a branch-and-bound algorithm for the global solution of NLPs; this NLP solution technique is termed αBB. In later work, Floudas *et al.* demonstrated a method by which rigorous values for α may be computed via interval analysis, provided that the functional form of the algebraic optimisation problem is known explicitly as an elementary function and can be manipulated symbolically [Adjiman et al., 1998b,a].

Although the problem of global optimisation of NLPs have received considerable attention in the literature, not many computer implementation currently exists. For this reason, we have developed the global optimisation software package `NLPGLOB`. The aim of this report is to be a user's guide. However, it also gives a brief overview of the theoretical foundations of the αBB algorithm, as well as several examples, so that the code and its usage is more comprehensible for the user.

CHAPTER 1

The αBB global optimisation algorithm

The αBB algorithm operates within a branch-and-bound framework and is designed to solve nonconvex minimisation problems of generic type represented by the following formulation:

$$\min_{\boldsymbol{p}} \; f(\boldsymbol{p}) \tag{1.1}$$

$$\text{s.t. } g_j(\boldsymbol{p}) = 0\,, \quad j = 1 \ldots n_{ctre} \tag{1.2}$$

$$g_j(\boldsymbol{p}) \leq 0\,, \quad j = n_{ctre} + 1 \ldots n_{ctr} \tag{1.3}$$

$$\boldsymbol{p} \in \mathscr{P} \subseteq \mathbb{R}^{n_p} \tag{1.4}$$

where n_{ctr} and n_{ctre} denote the number of constraints and equality constraints respectively, f, $\boldsymbol{g}$ and $\boldsymbol{h}$ belong to $\mathcal{C}^2$, the set of twice-differentiable functions and $\boldsymbol{p}$ is a vector of size n_p.

The theoretical properties of the αBB algorithm guarantee that such a problem can be solved to global optimality with finite ε-convergence. Each iteration of the algorithm consists of a *branching* step and a *bounding* step. In the latter, a lower bound is obtained by constructing valid convex underestimators for the functions in the problem and solving the resulting convex NLP to global optimality. An upper bound can be calculated by solving the original problem locally over each subdomain of the solution space.

The identification of the global optimal solution relies on the validity of the lower bounding problem, as well as the construction of increasingly tight lower bounding problems for successive partitions of the solution space. Such properties lead to the generation of a *non-decreasing sequence* of lower bounds which progresses towards the optimal solution. As can be expected, the convergence characteristics of the αBB algorithm are therefore significantly affected by the quality of the underestimator used.

A determining step in the convexification strategy is the decomposition of each nonlinear function into a sum of terms belonging to one or several categories: linear, bilinear, trilinear, fractional, fractional trilinear, convex, univariate concave or general nonconvex. Based on these categories, a function $f(\boldsymbol{p})$ with continuous second-order derivatives can be written as:

$$\begin{aligned} f(\boldsymbol{p}) &= LT(\boldsymbol{p}) + CT(\boldsymbol{p}) + \sum_{i=1}^{n_{bt}} b_i p_{B_i,1} p_{B_i,2} + \sum_{i=1}^{n_{tt}} t_i p_{T_i,1} p_{T_i,2} p_{T_i,3} \\ &+ \sum_{i=1}^{n_{ft}} f_i \frac{p_{F_i,1}}{p_{F_i,2}} + \sum_{i=1}^{n_{ftt}} ft_i \frac{p_{FT_i,1} p_{FT_i,2}}{p_{FT_i,3}} + \sum_{i=1}^{n_{ut}} UT_i(p_{UT_i}) + \sum_{i=1}^{n_{nt}} NT_i(\boldsymbol{p}) \end{aligned} \tag{1.5}$$

where $LT(\boldsymbol{p})$ is a linear term; $CT(\boldsymbol{p})$ is a convex term; n_{bt} is the number of bilinear terms, $p_{B_i,1}$ and $p_{B_i,2}$ denote the two variables that participate in the i-th bilinear term and b_i is its weighting coefficient; n_{tt} is the number of trilinear terms, $p_{T_i,1}$, $p_{T_i,2}$ and $p_{T_i,3}$ denote the three variables that participate in the i-th trilinear term and t_i is its weighting coefficient; n_{ft} is the number of fractional terms, $p_{F_i,1}$ and $p_{F_i,2}$ denote the two variables that participate in the i-th fractional term and f_i is its weighting coefficient; n_{ftt} is the number of fractional trilinear terms, $p_{FT_i,1}$, $p_{FT_i,2}$ and $p_{FT_i,3}$ denote the three variables that participate in the i-th fractional trilinear term and ft_i is its weighting coefficient; n_{ut} is the number of univariate concave terms, $UT_i(p_{UT_i})$ is the i-th univariate concave term and p_{UT_i} denotes the variable that participates in UT_i; n_{nt} is the number of general nonconvex terms and $NT_i(\boldsymbol{p})$ is the i-th general nonconvex term.

Techniques can be devised to generate valid and, in some cases, very tight convex underestimators of these terms. A detailed description of the treatment of such terms can be found in Androulakis et al. [1995] or Adjiman et al. [1998b]. In constructing a convex underestimator for the overall function, it is first noted that linear and convex terms do not require any transformation. The convex envelope of nonconvex terms of special structure (*i.e.* bilinear, trilinear, fractional, trilinear fractional and univariate concave terms) and general structure are detailed in the two subsequent paragraphs.

1.1 Underestimating nonconvex term of special structure

The convex envelope of bilinear, trilinear, fractional, trilinear fractional and univariate concave terms can be constructed by the following simple rules.

Underestimating bilinear terms

In the case of a bilinear term xy, a tight convex lower bound over the domain $\left[x^L, x^U\right] \times \left[y^L, y^U\right]$ is obtained by introducing a new variable ω_B which replaces every occurrence of xy in the problem and adding the four following linear inequality constraints:

$$\omega_B \geq x^L y + x y^L - x^L y^L \tag{1.6a}$$
$$\omega_B \geq x^U y + x y^U - x^U y^U \tag{1.6b}$$
$$\omega_B \leq x^U y + x y^L - x^U y^L \tag{1.6c}$$
$$\omega_B \leq x^L y + x y^U - x^L y^U \tag{1.6d}$$

Underestimating trilinear terms

A trilinear term of the form xyz can be underestimated by introducing and bounding a new variable ω_T by the following eight linear inequality constraints:

$$\omega_T \geq xy^Lz^L + x^Lyz^L + x^Ly^Lz - 2x^Ly^Lz^L \tag{1.7a}$$
$$\omega_T \geq xy^Uz^U + x^Uyz^L + x^Uy^Lz - x^Uy^Lz^L - x^Uy^Uz^U \tag{1.7b}$$
$$\omega_T \geq xy^Lz^L + x^Lyz^U + x^Ly^Uz - x^Ly^Uz^U - x^Ly^Lz^L \tag{1.7c}$$
$$\omega_T \geq xy^Uz^L + x^Uyz^U + x^Ly^Uz - x^Ly^Uz^L - x^Uy^Uz^U \tag{1.7d}$$
$$\omega_T \geq xy^Lz^U + x^Lyz^L + x^Uy^Lz - x^Uy^Lz^U - x^Ly^Lz^L \tag{1.7e}$$
$$\omega_T \geq xy^Lz^U + x^Lyz^U + x^Uy^Uz - x^Ly^Lz^U - x^Uy^Uz^U \tag{1.7f}$$
$$\omega_T \geq xy^Uz^L + x^Uyz^L + x^Ly^Lz - x^Uy^Uz^L - x^Ly^Lz^L \tag{1.7g}$$
$$\omega_T \geq xy^Uz^U + x^Uyz^U + x^Uy^Uz - 2x^Uy^Uz^U \tag{1.7h}$$

Underestimating fractional terms

Fractional terms of the form $\frac{x}{y}$ are underestimated by introducing a new variable ω_F and two new inequality constraints which depend on the sign of the bounds on x:

$$\omega_T \geq \begin{cases} \frac{x^L}{y} + \frac{x}{y^U} - \frac{x^L}{y^U} & \text{if } x^L \geq 0 \\ -\frac{x}{y^U} - \frac{x^Ly}{y^Ly^U} + \frac{x^L}{y^L} & \text{if } x^L < 0 \end{cases} \tag{1.8a}$$
$$\omega_T \geq \begin{cases} \frac{x^U}{y} + \frac{x}{y^L} - \frac{x^U}{y^L} & \text{if } x^U \geq 0 \\ -\frac{x}{y^L} - \frac{x^Uy}{y^Ly^U} + \frac{x^U}{y^U} & \text{if } x^U < 0 \end{cases} \tag{1.8b}$$

Underestimating fractional trilinear terms

For fractional trilinear term of the form $\frac{xy}{z}$, an underestimator is derived by introducing and bounding a new variable ω_{FT} by the following eight inequality constraints for x^L, y^L, $z^L \geq 0$:

$$\omega_{FT} \geq \frac{xy^L}{z^U} + \frac{x^Ly}{z^U} + \frac{x^Ly^L}{z} - 2\frac{x^Ly^L}{z^U} \tag{1.9a}$$
$$\omega_{FT} \geq \frac{xy^L}{z^U} + \frac{x^Ly}{z^L} + \frac{x^Ly^U}{z} - \frac{x^Ly^U}{z^L} - \frac{x^Ly^L}{z^U} \tag{1.9b}$$
$$\omega_{FT} \geq \frac{xy^U}{z^L} + \frac{x^Uy}{z^U} + \frac{x^Uy^L}{z} - \frac{x^Uy^L}{z^U} - \frac{x^Uy^U}{z^L} \tag{1.9c}$$
$$\omega_{FT} \geq \frac{xy^U}{z^U} + \frac{x^Uy}{z^L} + \frac{x^Ly^U}{z} - \frac{x^Ly^U}{z^U} - \frac{x^Uy^U}{z^L} \tag{1.9d}$$
$$\omega_{FT} \geq \frac{xy^L}{z^U} + \frac{x^Ly}{z^L} + \frac{x^Uy^L}{z} - \frac{x^Uy^L}{z^L} - \frac{x^Ly^L}{z^U} \tag{1.9e}$$
$$\omega_{FT} \geq \frac{xy^U}{z^U} + \frac{x^Uy}{z^L} + \frac{x^Ly^U}{z} - \frac{x^Ly^U}{z^U} - \frac{x^Uy^U}{z^L} \tag{1.9f}$$
$$\omega_{FT} \geq \frac{xy^L}{z^U} + \frac{x^Ly}{z^L} + \frac{x^Uy^L}{z} - \frac{x^Uy^L}{z^L} - \frac{x^Ly^L}{z^U} \tag{1.9g}$$
$$\omega_{FT} \geq \frac{xy^U}{z^L} + \frac{x^Uy}{z^L} + \frac{x^Uy^U}{z} - 2\frac{x^Uy^U}{z^L} \tag{1.9h}$$

Underestimating univariate concave terms

Univariate concave terms are trivially underestimated by their linearisation at the lower bound of the variable range. Thus the convex envelope of the concave function $ut(p)$ over $\left[p^L, p^U\right]$ is the following linear function of p:

$$ut\left(p^L\right) + \frac{ut\left(p^U\right) - ut\left(p^L\right)}{p^U - p^L}\left(p - p^L\right) \tag{1.10}$$

The addition of no additional variable or constraint is therefore required to compute the best convex underestimator for a univariate concave term.

1.2 Underestimating nonconvex terms of general structure

For the most general nonconvexities, a function $f(\boldsymbol{p})$ is underestimated over the entire domain $\left[\boldsymbol{p}^L, \boldsymbol{p}^U\right]$ by the function $\mathcal{L}(\boldsymbol{p})$ defined as:

$$\mathcal{L}(\boldsymbol{p}) = f(\boldsymbol{p}) + \sum_{i=1}^{n_p} \alpha_i \left(p_i^L - p_i\right)\left(p_i^U - p_i\right) \tag{1.11}$$

where the α_i's are positive scalars.

Given sufficiently large values of the α_i parameters, all nonconvexities in the original function $f(\boldsymbol{p})$ can be overpowered by the convex quadratic term and $\mathcal{L}(\boldsymbol{p})$ is therefore a valid convex underestimator. Since $\mathcal{L}(\boldsymbol{p})$ is convex means that its Hessian matrix $H_{\mathcal{L}}(\boldsymbol{p})$ is positive semi-definite, a useful convexity condition is derived by noting that $H_{\mathcal{L}}(\boldsymbol{p})$ is related to the Hessian matrix $H_f(\boldsymbol{p})$ of $f(\boldsymbol{p})$ by:

$$H_{\mathcal{L}}(\boldsymbol{p}) = H_f(\boldsymbol{p}) + 2\Delta \tag{1.12}$$

where Δ is a diagonal matrix whose diagonal elements are the α_i's and is referred to as the *diagonal shift matrix*.

To derive a valid convex underestimator therefore consists in generating a set of α parameters satisfying the following theorem:

Theorem 1 *$\mathcal{L}(\boldsymbol{p})$, as defined in (1.11), is convex iff $H_f(\boldsymbol{p}) + 2\Delta = H_f(\boldsymbol{p}) + 2diag(\alpha_i)$ is positive semi-definite for all $\boldsymbol{p} \in \left[\boldsymbol{p}^L, \boldsymbol{p}^U\right]$.*

In recent years, a number of deterministic methods have been devised in order to automatically identify an appropriate diagonal shift matrix, based on interval arithmetic. Two classes of approaches to this problem are considered subsequently: (*i*) uniform diagonal shift of the Hessian matrix of $f(\boldsymbol{p})$; (*ii*) non-uniform diagonal shift matrix of the Hessian matrix of $f(\boldsymbol{p})$. For further details refer, for instance, to Adjiman et al. [1998b].

1.2.1 Uniform diagonal shift matrix

For this class of methods, the underestimator is re-formulated using a single α value:

$$\mathcal{L}(\boldsymbol{p}) = f(\boldsymbol{p}) + \alpha \sum_{i=1}^{n_p} \left(p_i^L - p_i\right)\left(p_i^U - p_i\right) \tag{1.13}$$

All non-zero elements of Δ are therefore equal to α. It can be shown that $\mathcal{L}(\boldsymbol{p})$ is a valid convex underestimator of $f(\boldsymbol{p})$ iff:

$$\alpha \;\geq\; \max\left\{0, -\frac{1}{2}\min_{i,\boldsymbol{p^L}\leq\boldsymbol{p}\leq\boldsymbol{p}^U}\lambda_i(\boldsymbol{p})\right\} \tag{1.14}$$

where the λ_i's are the eigenvalues of $H_f(\boldsymbol{p})$.

Considering a interval Hessian matrix $[H_f] \supseteq \left\{H_f(\boldsymbol{p}), \boldsymbol{p} \in \left[\boldsymbol{p}^L, \boldsymbol{p}^U\right]\right\}$, it can be shown that a sufficient condition is given by:

$$\alpha \;\geq\; \max\left\{0, -\frac{1}{2}\lambda_{\min}([H_f])\right\} \tag{1.15}$$

where $\lambda_{\min}([H_f])$ is the minimum eigenvalue of the interval matrix family $[H_f]$. Examples of such a class of methods are [Adjiman et al., 1998b]:

- Gerschgorin's theorem for interval matrices;
- E-matrix method;
- Mori and Kokane's method;
- lower bounding Hessian method.

1.2.2 Non-uniform diagonal shift matrix

This class of methods allows the calculation of a different α value for each variable in order to construct an underestimator of the form shown in equation (1.11). In this case, the non-zero elements of the diagonal shift matrix can no longer be related to the minimum eigenvalue of the interval Hessian matrix $[H_f]$.

Examples of such a class of methods are [Adjiman et al., 1998b]:

- scaled Gerschgorin's theorem for interval matrices;
- H-matrix method;
- minimisation of maximum separation distance method.

1.2.3 On the role of interval calculations

The tractability of all the aforementioned α calculation methods is ensured through the use of interval arithmetic to obtain bounds on the second-order derivatives. Regardless of the chosen procedure, the quality of the constructed underestimator depends, to a large extent, on the accuracy of the intervals generated for each Hessian matrix element. However, the tightest possible bounds cannot be derived for the general case. Wider but guaranteed intervals can anyway be generated using classical interval arithmetic techniques, but the final result may differ depending on the initial analytical representation of the matrix elements since natural interval extension evaluates each term independently.

1.3 Overall convex underestimator

Based on the decomposition approach given in equation (1.5) and every customised underestimators discussed in § 1.1 and § 1.2, the corresponding lower bounding function $\mathcal{L}(\boldsymbol{p})$ to $f(\boldsymbol{p})$ is:

$$
\begin{aligned}
f(\boldsymbol{p}) \quad = \quad & LT(\boldsymbol{p}) + CT(\boldsymbol{p}) + \sum_{i=1}^{n_{bt}} b_i \omega_{B_i} + \sum_{i=1}^{n_{tt}} t_i \omega_{T_i} + \sum_{i=1}^{n_{ft}} f_i \omega_{F_i} + \sum_{i=1}^{n_{ftt}} ft_i \omega_{FT_i} \qquad (1.16) \\
& + \sum_{i=1}^{n_{ut}} \left(UT_i\left(p^L_{UT_i}\right) + \frac{UT_i\left(p^U_{UT_i}\right) - UT_i\left(p^L_{UT_i}\right)}{p^U_{UT_i} - p^L_{UT_i}} \left(p_i - p^L_{UT_i}\right) \right) \\
& + \sum_{i=1}^{n_{nt}} \left(NT_i(\boldsymbol{p}) + \sum_{j=1}^{n_p} \alpha_{ij} \left(p^L_j - p_j\right)\left(p^U_j - p_j\right) \right)
\end{aligned}
$$

where α_{ij} corresponds to the i-th general nonconvex term and the j-th variable; the ω_{B_i}, ω_{T_i}, ω_{F_i} and ω_{FT_i} variables must satisfy constraints of the forms given by the sets of equation (1.6), (1.7), (1.8) and (1.9) respectively.

Every customised underestimator is a function of the size of the domain under consideration. Since the αBB algorithm follows a branch-and-bound approach, this domain is systematically reduced at each new node of the tree so that tighter lower bounding functions can be generated through updates of equations (1.6) to (1.11). Thus the lower bounds on the problem form a non-decreasing sequence, as required for the identification of the global optimum solution. In addition, the quality of the variable bounds provided by the user can be expected to greatly influence the convergence of the algorithm and the investment of computational time in the calculation of tighter bounds is likely to result in improved performance. This fact will be discussed later in Chapter 2.

From a general point of view, any twice-differentiable nonconvex function can be treated as a single nonconvex term. However, the decomposition of functions into terms can improve the performance of the αBB algorithm in two ways: (i) the use of tight underestimators for certain types of terms increase the quality of the lower bounds generated for the optimum solution; (ii) the separation of large terms into terms involving a smaller number of variables results in a decrease of computational expense.

1.4 Equality constraints

In order to generate a valid lower bound on the global solution of the nonconvex NLP, the underestimating NLP generated in each subdomain must be convex. This imply that all inequality constraints in the lower bounding problem should be convex, all equality constraints should be linear and that the size of the feasible region must be increased relative to that of the original nonconvex problem.

Strategies that can be used to underestimate a nonlinear equality constraint depend on the type of terms it involves.

- For equalities in which only linear, bilinear, fractional and fractional trilinear terms appear, the nonlinear terms are replaced by new variables which participate linearly in the problem. The equality resulting from the substitution is therefore linear.

Moreover, since the set of values these new variables can take on is a superset of the values that can be attained by the nonlinear term, the linear equality corresponds to an enlarged feasible region. Thus, given the equality:

$$\begin{aligned} 0 \quad = \quad & LT(\boldsymbol{p}) + \sum_{i=1}^{n_{bt}} b_i p_{B_i,1} p_{B_i,2} + \sum_{i=1}^{n_{tt}} t_i p_{T_i,1} p_{T_i,2} p_{T_i,3} \\ & + \sum_{i=1}^{n_{bt}} f_i \frac{p_{F_i,1}}{p_{F_i,2}} + \sum_{i=1}^{n_{ftt}} ft_i \frac{p_{FT_i,1} p_{FT_i,2}}{p_{FT_i,3}} \end{aligned} \tag{1.17}$$

the following underestimator can be used:

$$0 \quad = \quad LT(\boldsymbol{p}) + \sum_{i=1}^{n_{bt}} b_i \omega_{B_i} + \sum_{i=1}^{n_{tt}} t_i \omega_{T_i} + \sum_{i=1}^{n_{ft}} f_i \omega_{F_i} + \sum_{i=1}^{n_{ftt}} ft_i \omega_{FT_i} \tag{1.18}$$

where the notation is as previously defined, and the appropriate inequality constraint for the $\boldsymbol{\omega}$ variables are added to the problem.

- For nonlinear equality constraint containing convex or general nonconvex terms, the equality obtained by simple substitution of the corresponding underestimator is nonlinear. Furthermore, if it contains univariate concave terms, it is linear but corresponds to a different feasible region. In the presence of either convex, univariate concave or general nonconvex terms, the original equality $g_j(\boldsymbol{p}) = 0$ must therefore be split as two inequalities of opposite sign:

$$\left\{ \begin{array}{rcl} g_j(\boldsymbol{p}) & \leq & 0 \\ -g_j(\boldsymbol{p}) & \leq & 0 \end{array} \right.$$

  These two inequality constraints must then be underestimated independently. The univariate concave terms appearing in the nonconvex equality become convex terms in one of the two inequalities while the convex terms become concave, and the general nonconvex terms become either concave or remain nonconvex.

1.5 Illustrative example (Example 1)

The illustrating example considered in this paragraph is a reactor network design problem, taken from Ryoo and Sahinidis [1995] and illustrated in Fig. 1.1, which is known to have caused difficulties for other global optimisation methods.

This problem is a nonlinearly constrained nonlinear optimisation problem of the following form:

$$\begin{array}{lrcl} & \multicolumn{3}{c}{\min_{\boldsymbol{p}} \; -p_4} \\ \text{s.t.} & p_1 + k_1 p_1 p_5 & = & 1 \\ & p_2 - p_1 + k_2 p_2 p_6 & = & 0 \\ & p_3 + p_1 + k_3 p_3 p_5 & = & 1 \\ & p_4 - p_3 + p_2 - p_1 + k_4 p_4 p_6 & = & 0 \\ & \sqrt{p_5} + \sqrt{p_6} & \leq & 4 \\ & (p_1, p_2, p_3, p_4, p_5, p_6) & \geq & (0, 0, 0, 0, 10^{-5}, 10^{-5}) \\ & (p_1, p_2, p_3, p_4, p_5, p_6) & \leq & (1, 1, 1, 1, 16, 16) \end{array} \tag{NLP1}$$

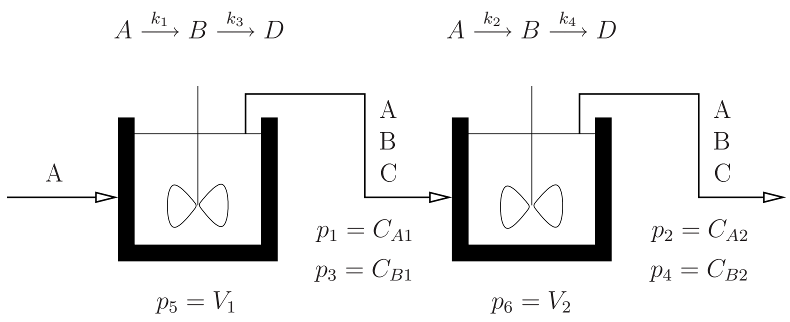

Figure 1.1: Reactor network design problem.

where k_1, k_2, k_3 and k_4 are parameters defined as $k_1 = 0.09755988$, $k_2 = 0.99k_2$, $k_3 = 0.0391908$ and $k_4 = 0.9k_3$.

The problem (NLP1) contains both univariate concave terms and bilinear terms. Therefore, the lower bounding problem can be constructed in two ways.

Case 1 by underestimating linearly each bilinear and univariate concave terms:

$$
\begin{array}{lrcl}
 & \min_{\boldsymbol{p},\boldsymbol{\omega}} \; -p_4 & & \\
\text{s.t.} & p_1 + k_1\omega_1 & = & 1 \\
 & p_2 - p_1 + k_2\omega_2 & = & 0 \\
 & p_3 + p_1 + k_3\omega_3 & = & 1 \\
 & p_4 - p_3 + p_2 - p_1 + k_4\omega_4 & = & 0 \\
 & \displaystyle\sum_{i=5}^{6}\left(\sqrt{p_5^L} + \frac{\sqrt{p_5^U} - \sqrt{p_5^L}}{p_5^U - p_5^U}\left(p_5 - p_5^L\right)\right) & \leq & 4 \\
 & p_1^L p_5 + p_1 p_5^L - p_1^L p_5^L & \leq & \omega_1 \\
 & p_1^U p_5 + p_1 p_5^U - p_1^U p_5^U & \leq & \omega_1 \\
 & p_1^U p_5 + p_1 p_5^L - p_1^U p_5^L & \geq & \omega_1 \\
 & p_1^L p_5 + p_1 p_5^U - p_1^L p_5^U & \geq & \omega_1 \\
 & p_2^L p_6 + p_2 p_6^L - p_2^L p_6^L & \leq & \omega_2 \\
 & p_2^U p_6 + p_2 p_6^U - p_2^U p_6^U & \leq & \omega_2 \\
 & p_2^U p_6 + p_2 p_6^L - p_2^U p_6^L & \geq & \omega_2 \\
 & p_2^L p_6 + p_2 p_6^U - p_2^L p_6^U & \geq & \omega_2 \\
 & p_3^L p_5 + p_3 p_5^L - p_3^L p_5^L & \leq & \omega_3 \\
 & p_3^U p_5 + p_3 p_5^U - p_3^U p_5^U & \leq & \omega_3 \\
 & p_3^U p_5 + p_3 p_5^L - p_3^U p_5^L & \geq & \omega_3 \\
 & p_3^L p_5 + p_3 p_5^U - p_3^L p_5^U & \geq & \omega_3 \\
 & p_4^L p_6 + p_4 p_6^L - p_4^L p_6^L & \leq & \omega_4 \\
 & p_4^U p_6 + p_4 p_6^U - p_4^U p_6^U & \leq & \omega_4 \\
 & p_4^U p_6 + p_4 p_6^L - p_4^U p_6^L & \geq & \omega_4 \\
 & p_4^L p_6 + p_4 p_6^U - p_4^L p_6^U & \geq & \omega_4 \\
 & \boldsymbol{p}^L \leq \boldsymbol{p} & \leq & \boldsymbol{p}^U
\end{array}
\tag{LBP1}
$$

Case 2 by considering these terms as general twice-differentiable nonconvex terms (LBP2).

$$
\begin{array}{llcl}
 & \min_{\boldsymbol{p}} \; -p_4 & & \\
\text{s.t.} & p_1 + k_1 p_1 p_5 + \alpha_{1,1}\left(p_1^L - p_1\right)\left(p_1^U - p_1\right) & & \\
 & \quad +\alpha_{1,5}\left(p_5^L - p_5\right)\left(p_5^U - p_5\right) & \leq & 1 \\
 & -p_1 - k_1 p_1 p_5 + \alpha_{2,1}\left(p_1^L - p_1\right)\left(p_1^U - p_1\right) & & \\
 & \quad +\alpha_{2,5}\left(p_5^L - p_5\right)\left(p_5^U - p_5\right) & \leq & -1 \\
 & p_2 - p_1 + k_2 p_2 p_6 + \alpha_{3,2}\left(p_2^L - p_2\right)\left(p_2^U - p_2\right) & & \\
 & \quad +\alpha_{3,6}\left(p_6^L - p_6\right)\left(p_6^U - p_6\right) & \leq & 0 \\
 & -p_2 + p_1 - k_2 p_2 p_6 + \alpha_{4,2}\left(p_2^L - p_2\right)\left(p_2^U - p_2\right) & & \\
 & \quad +\alpha_{4,6}\left(p_6^L - p_6\right)\left(p_6^U - p_6\right) & \leq & 0 \\
 & p_3 + p_1 + k_3 p_3 p_5 + \alpha_{5,3}\left(p_3^L - p_3\right)\left(p_3^U - p_3\right) & & \\
 & \quad +\alpha_{5,5}\left(p_5^L - p_5\right)\left(p_5^U - p_5\right) & \leq & 1 \\
 & -p_3 - p_1 - k_3 p_3 p_5 + \alpha_{6,3}\left(p_3^L - p_3\right)\left(p_3^U - p_3\right) & & \\
 & \quad +\alpha_{6,5}\left(p_5^L - p_5\right)\left(p_5^U - p_5\right) & \leq & -1 \\
 & p_4 - p_3 + p_2 - p_1 + k_4 p_4 p_6 + \alpha_{7,4}\left(p_4^L - p_4\right)\left(p_4^U - p_4\right) & & \\
 & \quad +\alpha_{7,6}\left(p_6^L - p_6\right)\left(p_6^U - p_6\right) & \leq & 0 \\
 & -p_4 + p_3 - p_2 + p_1 - k_4 p_4 p_6 + \alpha_{8,4}\left(p_4^L - p_4\right)\left(p_4^U - p_4\right) & & \\
 & \quad +\alpha_{8,6}\left(p_6^L - p_6\right)\left(p_6^U - p_6\right) & \leq & 0 \\
 & \sqrt{p_5} + \sqrt{p_6} + \alpha_{9,5}\left(p_5^L - p_5\right)\left(p_5^U - p_5\right) & & \\
 & \quad +\alpha_{9,6}\left(p_6^L - p_6\right)\left(p_6^U - p_6\right) & \leq & 4 \\
 & \boldsymbol{p}^L \leq \boldsymbol{p} & \leq & \boldsymbol{p}^U
\end{array}
\tag{LBP2}
$$

where the shift parameters α_{ij}'s are computed using interval arithmetic techniques. Concerning the nonconvex term $k_1 p_1 p_5$, for instance:

$$
[H_1] = \begin{bmatrix} 0 & 0 & 0 & 0 & k_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ k_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{1.19}
$$

Thus, the application of the scaled Gerschgorin's theorem gives:

$$
\boldsymbol{\alpha}_1 = \left[\frac{k_1}{2} \; 0 \; 0 \; 0 \; \frac{k_1}{2} \; 0\right]^T \tag{1.20}
$$

Note that in both cases, the resulting lower bounding problems (LBP1) and (LBP2) are linear problems (LP).

CHAPTER 2

Algorithmic issues

Based on the theoretical material described is Chapter 1, an implementation of the algorithm was developed. The main objective was to create an efficient optimisation environment for NLP problems involving twice-differentiable functions. The package `NLPGLOB` implements the αBB algorithm in `Fortran77` and possesses the following features:

- several alternatives for the construction of the lower bounding problem, depending on the mathematical characteristics of the terms appearing in the problem;
- several methods to compute shift parameters α;
- several local NLP solvers;
- several branching strategies;
- several variable bounds update strategies.

The structure of the algorithm is illustrated in Fig. 2.1. The following discussion outlines the branching and variable bounding rules which are used in order to enhance the performances of the αBB algorithm.

2.1 Branching strategies

Although it does not present any theoretical difficulties, the branching step of any branch-and-bound algorithm often has significant effect on the rate of convergence. This is especially true for the αBB algorithm since the quality of the underestimator depends on the variable bounds in a variety of ways. For instance, if a variable participates only linearly in the problem, branching on it will not have any effect on the accuracy of the convex lower bounding functions. On the other hand, reducing the range of a variable raised to high power is likely to result in much tighter underestimating problems. To take advantage of these observations, the implementation of the αBB algorithm offers some choice in the branching strategies. Four alternatives are currently available in the package `NLPGLOB`:

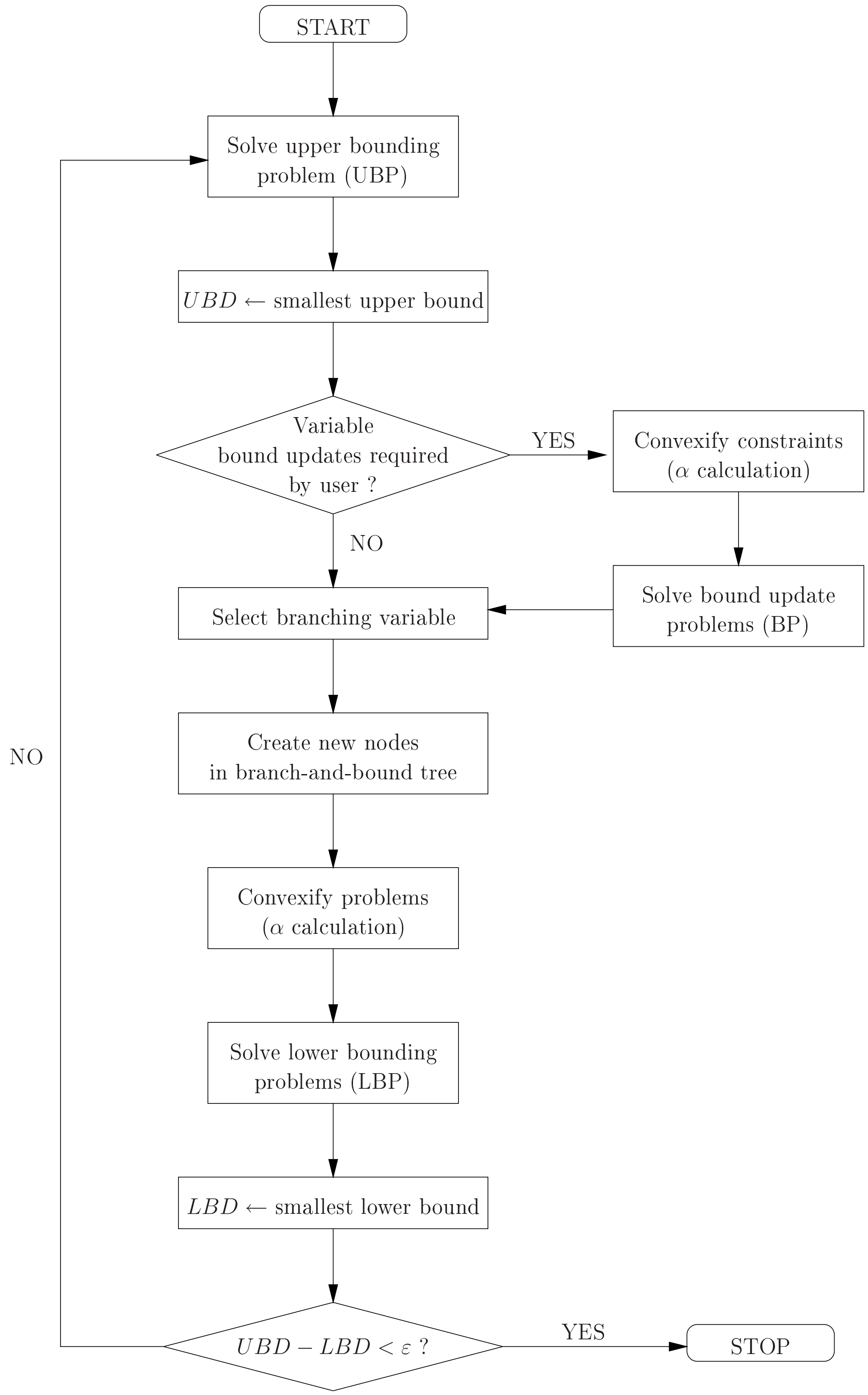

Figure 2.1: Flowchart for the αBB algorithm (`NLPGLOB`).

Strategy 1 Use a k-section on all or some of the variables;

Strategy 2 Use a measure of the quality of each term's underestimator, based on the maximum separation distance between a term and its corresponding underestimator;

Strategy 3 Use a measure of the quality of each term's underestimator, based on the separation distance at the optimum point;

Strategy 4 Use a measure of the overall influence of each variable on the quality of the lower bounding problem.

These branching strategies are briefly detailed below. Note that branching strategies 2, 3 and 4 are particularly effective since they take into account the sensitivity of the underestimators to the bounds used for each variable.

2.1.1 Strategy 1

By default, bisection on all variables is used. The user can however specify a restricted list of variables on which to branch. The interval to be bisected is determined through the *least reduced axis* rule whose application involves the calculation of a *current range* to *original range* ratio, denoted r_i, for each variable:

$$r_i = \frac{p_i^U - p_i^L}{p_{i,0}^U - p_{i,0}^L} \tag{2.1}$$

where $p_{i,0}^U$ and $p_{i,0}^L$ are respectively the upper and lower bounds on variable p_i at the first node of the branch-and-bound tree, and p_i^U and p_i^L are respectively the upper and lower bounds on variable p_i at the current node of the tree. The variable with the largest r_i is selected for branching.

2.1.2 Strategy 2

While the second branching option requires additional computational effort, it results in a significant improvement of the convergence time for difficult problems. A different measure μ is defined for each type of underestimator to facilitate the assessment of the quality of the lower bounding function [Maranas and Floudas, 1994, Androulakis et al., 1995, Adjiman et al., 1998a]. The maximum separation distance between the underestimator and the actual term at the optimal solution of the lower bound problem is one possible indicator of the degree of accuracy achieved in the construction of the convex approximation.

- For a bilinear term xy the maximum separation distance is:
$$\mu_B = \frac{\left(x^U - x^L\right)\left(y^U - y^L\right)}{4} \tag{2.2}$$
- For a univariate concave term $ut\left(\boldsymbol{x}\right)$, the maximum separation distance can be expressed as an convex optimisation problem so that μ_{UT} is given by:
$$\mu_{UT} = -\min_{x^L \le x \le x^U} \left\{-ut\left(x\right) + ut^L\left(x\right)\right\} \tag{2.3}$$
where $ut^L\left(x\right)$ is the linearisation of $ut\left(x\right)$ around x^L.

- For a general nonconvex term, the maximum separation distance μ_{NT} is:

$$\mu_{NT} = \frac{1}{4}\sum_{i=1}^{n_p} \alpha_i \left(x_i^U - x_i^L\right)^2 \tag{2.4}$$

where the α_i's are defined by equation (1.11).

Given a node to be partitioned, the values of μ_B, μ_{UT} and μ_{NT} are calculated for each term in accordance with its type. The term which appears to have the worst underestimator, *i.e.* the largest μ, is then used as a basis for the selection of the branching variables. Out of the set of variables that participate in that term, the one with the *least reduced axis*, as defined by equation (2.1), is chosen for k-section. Under this strategy, the influence of the variable bounds on the quality of the underestimators is directly taken into account, and hence this adaptive branching scheme ensures the effective tightening of the lower bounding problem from iteration to iteration.

2.1.3 Strategy 3

This strategy is a variant of Strategy 2. Instead of computing the maximum separation distance between a term and its underestimator, their separation distance at the *optimum solution* of the lower bounding problem is used [Adjiman et al., 1998a]:

- The bilinear term xy measure is:

$$\mu_B = |x^\star y^\star - \omega^\star| \tag{2.5}$$

where ω is the variable that is substituted for the bilinear term xy in order to construct its convex envelope and the * superscript denotes the value of the variable at the solution of the current lower bounding problem.

- The measures μ_T, μ_F and μ_{FT}, for trilinear, fractional and fractional trilinear terms, are obtained in a similar fashion.

- The measure for univariate concave term $ut\left(\boldsymbol{x}\right)$ is:

$$\mu_{UT} = ut\left(x^\star\right) - ut^L\left(x^\star\right) \tag{2.6}$$

- Finally, the general nonconvex term measure is:

$$\mu_{NT} = \sum_{i=1}^{n_p} \alpha_i \left(p_i^U - p_i^\star\right)\left(p_i^L - p_i^\star\right) \tag{2.7}$$

2.1.4 Strategy 4

The fourth branching procedure takes the approach of Strategy 3 one step further by considering the overall influence of each variable on the convex problem [Adjiman et al., 1998a]. After the relevant measures μ_B, μ_T, μ_F, μ_{FT}, μ_{UT} and μ_{NT} have been calculated for every term, a measure μ_p of each variable's contribution may be obtained as follow:

$$\mu_{p_i} = \sum_{j\in BT_i} \mu_B^j + \sum_{j\in TT_i} \mu_T^j + \sum_{j\in FT_i} \mu_F^j + \sum_{j\in FTT_i} \mu_{FT}^j + \sum_{j\in UT_i} \mu_{UT}^j + \sum_{j\in NT_i} \mu_{NT}^j \tag{2.8}$$

where μ_{p_i} is the measure of the i-th variable; BT_i is the index set of the bilinear terms in which the i-th variable participates and μ_{BT}^j is the measure of the j-th bilinear term; TT_i is the index set of the trilinear terms in which the i-th variable participates and μ_{TT}^j is the measure of the j-th trilinear term; FT_i is the index set of the fractional terms in which the i-th variable participates and μ_{FT}^j is the measure of the j-th fractional term; FTT_i is the index set of the fractional trilinear terms in which the i-th variable participates and μ_{FTT}^j is the measure of the j-th fractional trilinear term; UT_i is the index set of the univariate concave terms in which the i-th variable participates and μ_{UT}^j is the measure of the j-th univariate concave term; NT_i is the index set of the general nonconvex terms in which the i-th variable participates and μ_{NT}^j is the measure of the j-th general nonconvex term;

The variable with the largest measure μ_p is selected as the branching variable, and k-section can be performed on it. If two or more variables have the same measure, the *least reduced axis* test is performed to distinguish them.

2.2 Variable bound updates

The quality of the convex lower bounding problem can also be improved by ensuring that the variable bounds are as tight as possible. In the current implementation of the αBB algorithm, variable bound updates can either be performed at the onset of an αBB run or at each iteration.

In both cases, the same procedure is followed in order to construct the bound update problem. Given a solution domain, the convex underestimator for every constraint in the original NLP is formulated. The *bound problems* for variable p_i are then expressed as:

$$p_i^{L,\text{new}} = \left\{ \begin{array}{ll} \min_{\boldsymbol{p}} & p_i \\ \text{s.t.} & \mathcal{L}_g(\boldsymbol{p}) \leq \mathbf{0} \\ & \mathcal{L}_f(\boldsymbol{p}) \leq UBD \\ & \boldsymbol{p}^L \leq \boldsymbol{p} \leq \boldsymbol{p}^U \end{array} \right. \tag{BPL}$$

$$p_i^{U,\text{new}} = \left\{ \begin{array}{ll} \min_{\boldsymbol{p}} & -p_i \\ \text{s.t.} & \mathcal{L}_g(\boldsymbol{p}) \leq \mathbf{0} \\ & \mathcal{L}_f(\boldsymbol{p}) \leq UBD \\ & \boldsymbol{p}^L \leq \boldsymbol{p} \leq \boldsymbol{p}^U \end{array} \right. \tag{BPU}$$

where $\mathcal{L}_f(\boldsymbol{p})$ and $\mathcal{L}_g(\boldsymbol{p})$ are the convex underestimators of the objective function and of the constraints respectively, $\boldsymbol{p}^L$ and $\boldsymbol{p}^U$ are the best calculated bounds on the variables, and UBD is the current best upper bound on the global solution. Once a new lower bound $p_i^{L,\text{new}}$ on p_i has been computed via problem (BPL), this value is used in the formulation of problem (BPU) for the generation of an upper bound $p_i^{U,\text{new}}$.

Note that because of the computational expense incurred by an update of the bounds on all variables, it is often desirable to define a smaller subset of the variables on which this operation is to be performed. The criterion devised for the selection of the branching variables can be used in this instance since it provides a measure of the sensitivity of the problem to each variable. In the package `NLPGLOB` in particular, bounds update are carried out only for a fraction of the variables with a non-zero μ_p, as calculated in equation (2.8).

2.3 Future improvements of the package

Several improvements of the algorithm are currently under investigation. Their main objectives are: (*i*) to create a user interface that would be simple and flexible; (*ii*) to derive gradients and Hessian matrices via automatic differentiation, to generate interval Hessian matrices and to perform interval arithmetic automatically; (*iii*) to implement robust large-scale NLP solvers, such as rSQP [Ternet and Biegler, 1999], so that nonconvex problems with many parameters and/or constraints can be handled.

CHAPTER 3

Specification of the subroutines

The package NLPGLOB has to communicate with several subroutines. In addition to the principal routine NLPGLOB, subroutines containing information about the objective and constraint functions, their gradients and their structures (*i.e.* the specification of each nonconvex terms of special or general structure) have to be specified. Finally, the user can also choose among the local NLP solvers that are supported.

3.1 Subroutine NLPGLOB

Only a double-precision of the package NLPGLOB is currently available. The calling sequence is as follow :

```
      call NLPGLOB(np, nps, nctr, nctrs, nctre, nnconv, iwnconv, nspec,
     &             iwspec, rwspec, nconc, iwconc, pl, pu, linfo, info,
     &             lrnfo, rnfo, lrw, rw, liw, iw, llw, lw, rpar, ipar,
     &             func, grad, funcrel, gradrel, inthess, fconc)

      integer np, nps, nctr, nctrs, nctre, nnconv, iwnconv, nspec,
     &        iwspec, nconc, iwconc, linfo, info, lrnfo, lrw, liw, iw,
     &        llw, ipar
      logical lw
      double precision rwspec, pl, pu, rnfo, rw, rpar

      dimension pl(np), pu(np)
      dimension iwnconv(nnconv+1, np+1), iwspec(nspec+1, np+2),
     &          rwspec(nspec+1), iwconc(nconc+1, 2)
      dimension rw(lrwork), iw(liwork), lw(llwork)
      dimension info(linfo), rnfo(lrnfo), rpar(*), ipar(*)

      external func,grad,funcrel,gradrel,inthess,fconc
```

The specification of subroutine NLPGLOB is as follows :

`np` **[input]** Number of parameters in the nonconvex problem.

`nps` **[input]** Number (possibly zero) of additional parameters in the convex underestimated problem, due to nonconvex terms of special structure.

`nctr` **[input]** Number (possibly zero) of constraints in the nonconvex problem.

`nctrs` **[input]** Number (possibly zero) of additional constraints in the convex underestimated problem, due to nonconvex terms of special structure.

`nctre` **[input]** Number (possibly zero) of **linear** constraints in the nonconvex problem.

`nnconv` **[input]** Number (possibly zero) of twice-differentiable nonconvex terms of general structure.

`iwnconv(nnconv+1,np+1)` **[input]** Characterisation of each general twice-differentiable nonconvex term: only if `info(4)=1...3` ; otherwise, `iwnconv` can be defined as a dummy argument. Recall that the interval Hessian matrix of each general nonconvex term must be specified as well (see the detailed specification given in § 3.2.5).

`iwnconv(1...nnconv,1)` column indicating the function that contains each general nonconvex term:
`iwnconv(i,1)=0` : the i-th general nonconvex term appears in the objective function;
`iwnconv(i,1)=k` : the i-th general nonconvex term appears in the k-th constraint.

`iwnconv(1...nnconv,2...np+1)` columns indicating the participating variables in each general nonconvex term:
`iwnconv(i,j+1)=0` : the j-th parameter does not appear in the i-th general nonconvex term;
`iwnconv(i,j+1)=1` : the j-th parameter appears in the i-th general nonconvex term.

`nspec` **[input]** Number (possibly zero) of nonconvex terms of special structure ; special terms currently implemented are bilinear, fractional, trilinear and trilinear fractional terms.

`iwspec(nspec+1,np+2)` **[input]** Characterisation of each general nonconvex term of special structure: only if `info(4)=1...3` ; otherwise, `iwspec` can be defined as a dummy argument.

`iwspec(1...nspec,1)` column indicating the function that contains each term of special structure:
`iwspec(i,1)=0` : the i-th term appears in the objective function;
`iwspec(i,1)=k` : the i-th term appears in the k-th constraint.

`iwspec(1...nspec,2...np+1)` columns indicating the participating variables in each term of special structure:
`iwspec(i,j+1)=0` : the j-th parameter does not appear in the i-th term;
`iwspec(i,j+1)=1` : the j-th parameter appears in the numerator of the i-th term.

`iwspec(i,j+1)=-1` : the j-th parameter appears in the denominator of the i-th term.

`iwspec(1...nspec,np+2)` column indicating the index of the additional variable used to relax each special term:
`iwspec(i,np+2)=k` : the i-th special term is relaxed by the use of the additional parameter $k = 1 \dots n_{ps}$.

`rwspec(nspec+1)` **[input]** Specification of the weighting coefficient of each nonconvex term of special structure: only if `info(4)=1...3` ; otherwise, `rwspec` can be defined as a dummy argument.

`nconc` **[input]** Number (possibly zero) of unvariate concave terms.

`iwconc(nconc+1,2)` **[input]** Characterisation of each univariate concave term: only if `info(4)=1...3` ; otherwise, `iwconc` can be defined as a dummy argument. Recall that the expression of each univariate concave term must be defined as well (see the detailed specification given in § 3.2.6 below).

`iwconc(1...nconc,1)` column indicating the function that contains each univariate concave term:
`iwconc(i,1)=0` : the i-th univariate concave term appears in the objective function;
`iwconc(i,1)=k` : the i-th univariate concave term appears in the k-th constraint function.

`iwconc(1...nconc,2)` column indicating which parameter participates in each univariate concave term:
`iwconc(i,2)=k` : the k-th parameter participates in the i-th univariate concave term.

`pl(np)` **[input]** Lower bounds on the parameters.

`pu(np)` **[input]** Upper bounds on the parameters.

`lrnfo` **[input]** Length of real information vector `rnfo`. Should be at least 2.

`rnfo(lrnfo)` **[input]** Real information vector. The first 2 components should be either zero or positive. The number in parentheses indicates default value used if the corresponding entry is zero.

`rnfo(1)` relative convergence tolerance on the global optimum (=1d-3).

`rnfo(2)` absolute convergence tolerance of the local SQP solver (=1d-6).

`linfo` **[input]** Length of integer information vector `info`. Should be at least 14.

`info(linfo)` **[input/output]** Integer information vector. The first 14 components should be either zero, positive or negative. The number in parentheses indicates default value used if the corresponding entry is zero.

`info(1)` Communication with `NLPGLOB` (= 0).

[input]

`info(1)=-3` Display minimum needed workspace allocation.

`info(1)=-2` Check the analytical expressions provided in subprograms `grad` and `gradrel` with finite difference technique.

`info(1)=-1` Optimise the process by solving the NLP locally. An initial point p_0 **must** be provided from position `rw(1)` to `rw(np)` within real workspace vector `rw`. In this case, subroutines `funcrel`, `gradrel`, `inthess` and `fconc`, as well as arrays `iwnconv`, `iwspec` and `iwconc` can be defined as dummy arguments. Also, display options are set to `info(6)=3` and `info(12)=2`.

`info(1)=0` Perform global optimization.

[output]

`info(1)=0` Normal termination. A global optimum has been found within the tolerance given in `rnfo(1)`.

`info(1)=1` The actual solution is not contained within any node of the branch-and-bound tree ; the selected NLP solver may have failed to solve a feasible relaxed problem (check convergence tolerances).

`info(1)=2` No further node of the branch-and-bound tree to explore! The problem may be infeasible or the selected NLP solver may have failed to solve a feasible relaxed problem (check convergence tolerances).

`info(1)=3` The algorithm has been stopped after `maxit` (=`info(2)`) iterations.

`info(1)=4` There are wrong dimensions.

`info(1)>10` Values indicating failure ; specified more precisely in the selected NLP solver (`ifail`=`info(1)`-10).

`info(2)` [**input**] Maximum number of iterations in branch-and-bound algorithm (= 999).

`info(3)` [**input**] Choice of the method to compute shift parameters α_i ($i = 1 \ldots n_{nconv}$) for each general twice-differentiable nonconvex term (= 11). Currently implemented methods are:

uniform diagonal shift matrix

`info(3)=1` Gershgorin theorem;

`info(3)=2` E-matrix method;

`info(3)=3` Mori and Kokane's method;

`info(3)=4` Lower bounding hessian method

non-uniform diagonal shift matrix

`info(3)=11` Scaled Gershgorin theorem;

The interval Hessian matrices must be provided within subroutine `inthess` (see details in § 3.2.5 below). These matrices can be scaled by defining scaling coefficients from position `rw(6*np+3*nps+(nnconv+1)*(2*np**2+np)+1)` to `rw(6*np+3*nps+(nnconv+1)*(2*np**2+np)+np)` within real workspace vector `rw` –default values are equal to unity–.

If option `info(3)=2` is selected, the E-matrix can be defined row-wise from position `rw(7*np+3*nps+(nnconv+1)*(2*np**2+np)+1)` to `rw(7*np+3*nps+(nnconv+1)*(2*np**2+np)+np**2)` within real workspace vector `rw` –default values are equal to zero–.

`info(4)` [**input**] Specification of the branching strategy (= 0).

`info(4)=0` The interval to be bisected is determined among **all** parameters through the *least reduced axis* rule which application involves the calculation of a *current range to original range* ratio for each variable. The user can however specify a restricted list of variables on which to branch in the logical workspace vector `lw` from position `lw(1)` to `lw(np)`. In this case, arrays `iwnconv`, `iwspec`, `rwspec` and `iwconc`, as well as the function `fconc`, can be defined as dummy arguments.

`info(4)=1` The maximum separation distance between each underestimators and actual nonconvex terms are computed. Only the variables that participate in the term with the worst underestimator are considered for bisection using the *least reduced axis rule.*

`info(4)=2` Variant of `info(4)=1`. Separation distances are computed at the optimum solution value instead of using the maximum separation distance between the under- estimators and the related actual terms.

`info(4)=3` Extension of `info(4)=2`. The overall influence of each variable on the convex problem is considered by computing a measure of each variable contribution. The variable with the largest measure is then selected as the branching variable ; if more than two variables have the same measure, bisection is performed using the *least reduced axis rule.*

`info(5)` [**input**] Specification of the variable bound update strategy (= 0).

`info(5)=0` No variable bound update is performed.

`info(5)=1` Variable bound update is performed at the first iteration only.

`info(5)=2` Variable bound update is performed at each iteration.

`info(6)` [**input**] Specification of the desired level of information printed by the subroutine (= 2).

`info(6)=0` No output.

`info(6)=1` Only a final convergence analysis is given.

`info(6)=2` One line of intermediate results is printed at each iteration.

`info(6)=3` More detailed information is printed at each iteration, *i.e.* the solutions of each upper bound, lower bound and bound update problems.

`info(7)` [**input**] Specification of the desired output unit number `iout`, *i.e.* all write-statements start with the instruction `write(iout,...)` (= 6).

`info(8)` **[input]** Specification of the NLP solver (= 0).

- `info(8)=0` NLP solver `SLSQP` is used. `SLSQP` is a public domain code taken from `www.netlib.org` [Kraft, 1988].
- `info(8)=1` NLP solver `NLPQL` is used. `NLPQL` is a commercial SQP solver [Schittkowski, 1985] and cannot be included with the package (see website `www.uni-bayreuth.de/departments/math/~kschittkowski/nlpql.htm`). An empty subroutine is therefore provided that should be replaced by `NLPQL` source code if available.

`info(9)` **[input]** Specification of gradient computation (= 0). Subroutines `grad` and `gradrel` must be provided by the user. It is recommended to check the analytical gradient expressions by selecting `info(1)=-2`.

- `info(9)=0` Gradients computed via finite differences. In this case, subroutines `grad` and `gradrel` can be defined as dummy arguments.
- `info(9)=1` Gradients computed via analytic expressions.

`info(10)` **[input]** Specification of the maximum number of function evaluations calls during the line-search of the NLP solver (= 40).

`info(11)` **[input]** Specification of the maximum number of iterations of the NLP solver (= 999).

`info(12)` **[input]** Specification of the desired output level of the NLP solver (only if `info(6)=3`) (= 0).

- `info(12)=-1` Nothing is printed.
- `info(12)=0` Information is printed only if the NLP solver fails to compute the solution of a upper bound, lower bound or bound update problem.
- `info(12)=1` Only a final convergence analysis is given for each NLP sub-problem.
- `info(12)=2` One line of intermediate results is printed at each iteration of the NLP solver.
- `info(12)=3` More detailed information is printed at each iteration step, e.g. variable, constraint and multiplier values.
- `info(12)=4` In addition to `info(12)=3`, the NLP algorithm is stopped after each iteration of the NLP solver.

`info(13)` **[input]** Specification of the line-search method of the selected NLP solver.

Case `SLSQP` (=0)

- `info(13)=0` An exact line-search is performed.
- `info(13)<>0` The algorithm uses an Armijo-type line-search.

Case `NLPQL` (=0)

- `info(13)<0` The algorithm uses the $\ell 2$-augmented Lagrangian penalty function for the line-search.
- `info(13)>0` The $\ell 1$-penalty function is applied

`info(13)=0` info(13) $= 0$: the SQP solver starts with $\ell 2$-augmented Lagrangian penalty function and switches to the $\ell 1$-penalty function in error cases.

`info(14)` **[input]** Specification of the method used to solve QP subproblems: only if `NLPQL` is used (`info(8)=1`); no match if `SLSQP` is selected (`info(8)=0`) $(= 0)$.

`info(14)=0` The quadratic programming sub-problem is to be solved with a full positive definite quasi-Newton matrix.

`info(14)<>0` A Cholesky decomposition is performed and updated by `NLPQL`, so that the sub-problem matrix contains an upper triangular factor.

`lrw` **[input]** Dimension of the real workspace `rw`. A sufficiently large value should be given; if not enough, the program writes error message with the minimum necessary value.

`rw(lrw)` **[output]** Real workspace. See Appendix A for full description of the workspace organisation.

`liw` **[input]** Dimension of the integer workspace `iw`. A sufficiently large value should be given; if not enough, the program writes error message with the minimum necessary value.

`iw(liw)` **[output]** Integer workspace. See Appendix A for full description of the workspace organisation.

`llw` **[input]** Dimension of the logical workspace `lw`. A sufficiently large value should be given; if not enough, the program writes error message with the minimum necessary value.

`lw(llw)` **[output]** Logical workspace. See Appendix A for full description of the workspace organisation.

`rpar(*)` **[input/output]** User-defined real parameter vector workspace that is not altered by `NLPGLOB`. It can be used in communication among the user subroutines, but should not be a function of the optimised parameters `p`.

`ipar(*)` **[input/output]** User-defined integer parameter vector workspace that is not altered by `NLPGLOB`. It can be used in communication among the user subroutines, but should not be a function of the optimised parameters `p`.

`func` **[input]** Name of the user-defined subroutine that computes the value of the objective function and the constraints. This name must be declared in an `external` statement in the calling routine and passed as an argument to `NLPGLOB`. The detailed specification is given in § 3.2.1 below.

`grad` **[input]** Name of the user-defined subroutine that computes the gradients of the objective function and the constraints. This name must be declared in an `external` statement in the calling routine and passed as an argument to `NLPGLOB`. If gradients are computed by finite difference techniques (`info(9)=1`), this routine can be declared as a dummy argument. The detailed specification is given in § 3.2.2 below.

`funcrel` **[input]** Name of the user-defined subroutine that computes the value of the objective function and constraints of the convex underestimated problem. This name must be declared in an `external` statement in the calling routine and passed as an argument to `NLPGLOB`. The detailed specification is given in § 3.2.3 below.

`gradrel` **[input]** Name of the user-defined subroutine that computes the gradients of the objective function and constraints of the convex underestimated problem. This name must be declare din an `external` statement in the calling routine and passed as an argument to `NLPGLOB`. If gradients are computed by finite difference techniques (`info(9)=1`), this subroutine can be declared as a dummy argument. The detailed specification is given in § 3.2.4 below.

`inthess` **[input]** Name of the user-defined subroutine that computes the interval Hessian matrices of any general twice-differentiable nonconvex terms. This name must be declared in0 an `external` statement in the calling routine and passed as an argument to `NLPGLOB`. The detailed specification is given in § 3.2.5 below.

`fconc` **[input]** Name of the user-defined function that computes the values of any univariate concave term. This name must be declared in an `external` statement in the calling routine and passed as an argument to `NLPGLOB`. The detailed specification is given in § 3.2.6 below.

Illustrating example (Example 1 continued).

For the problem (NLP1), the dimensions are as follow; dimension of variables (6), number of constraints (5) and number of equality constraints (4).

In **Case 1**, the convex underestimated problem (LBP1) involves 4 additional variables and 8 additional inequality constraints. Sufficient dimensions of the workspace vectors have been found as `lrw=622656`, `liw=80073`, `llw=20065`.

The corresponding file is shown below.

```
      program Example1a

c     ------------------------------------------------------
c     This problem has been originally addressed by RYOO H.S.
c     and SAHINIDIS in "Global optimization of non-convex
c     NLPs and MINLPs with application in processes design",
c     Computers chem. Engng., 19:551-566, 1995 (example 20)
c
c     Reactor network design :
c
c                       min -x4
c
c          subject to :        x1-1+k1*x1*x5 = 0
c                             x2-x1+k2*x2*x6 = 0
c                           x3+x1-1+k3*x3*x5 = 0
c                       x4-x3+x2-x1+k4*x4*x6 = 0
c                           x5**0.5+x6**0.5 <= 4
```

```
c                          x1,x2,x3,x4,x5,x6 <= {1,1,1,1,16,16}
c                          x1,x2,x3,x4,x5,x6 >= {0,0,0,0,0,0}
c
c                           -------------
c
c                   k1    k3            k2    k4
c                 A --> B --> C      A --> B --> C
c
c          with :   k1 = 0.09755988
c                   k2 = 0.99*k1
c                   k3 = 0.0391908
c                   k4 = 0.9*k3
c
c     -------------------------------------------------------

        implicit none

c       NLPGLOB Arguments
        integer np, nps, nctr, nctrs, nctre, nnconv, iwnconv, nspec,
     &          iwspec, nconc, iwconc, lin, lrn, lrw, liw, llw, iw, ipar,
     &          info
        double precision rwspec, pl, pu, rw, rpar, rnfo
        logical lw
        external func, grad, funcrel, gradrel, hessint, fconc

c       Problem dimension
        parameter(np=6, nps=4, nctr=5, nctrs=16, nctre=4)
        dimension pl(np), pu(np)
        parameter (nnconv=0, nspec=4, nconc=2)
        dimension iwnconv(nnconv+1, np+1),iwspec(nspec+1, np+2),
     &            iwconc(nconc+1, 2),rwspec(nspec+1)

        parameter(lrw=622656, liw=80073, llw=20065)
        dimension rw(lrw), iw(liw), lw(llw)
        parameter(lin=14, lrn=2)
        dimension info(lin), rnfo(lrn), ipar(1), rpar(4)

c       aBB option/termination
        info(1) = 0
c       aBB maximal number of iterations
        info(2) = 9999
c       aBB alpha calculation
        info(3) = 11
c       aBB branching strategy
        info(4) = 2
c       aBB variable bound updates
        info(5) = 2
c       aBB display options
```

```
        info(6) = 2
c       aBB output unit
        info(7) = 6
c       NLP solver selection
        info(8) = 0
c       NLP solver gradient computation
        info(9) = 1
c       NLP solver maximal number of function evaluation during
c       linesearch (NLPQL only)
        info(10) = 99
c       NLP solver maximal number of iterations
        info(11) = 999
c       NLP solver display options
        info(12) = 0
c       NLP solver line-search option
        info(13) = 0
c       QP subproblem resolution option (NLPQL only)
        info(14) = 0
c       aBB relative convergence tolerance
        rnfo(1) = 1d-5
c       NLP solver absolute convergence tolerance
        rnfo(2) = 1d-10

c       Initialisation
        call INIT(np, nnconv, iwnconv, nspec, iwspec, rwspec, nconc,
     &            iwconc, lrw, rw, pl, pu, rpar, ipar)

c       Launch NLPGLOB
        call NLPGLOB(np, nps, nctr, nctrs, nctre, nnconv, iwnconv,
     &               nspec, iwspec, rwspec, nconc, iwconc, pl, pu, lin,
     &               info, lrn, rnfo, lrw, rw, liw, iw, llw, lw, rpar,
     &               ipar, func, grad, funcrel, gradrel, hessint, fconc)

      end
      subroutine INIT(np, nnconv, iwnconv, nspec, iwspec, rwspec, nconc,
     &                iwconc, lrw, rw, pl, pu, rpar, ipar)

        implicit none

c       Arguments
        integer np, nnconv, nspec, nconc, iwnconv, iwspec, iwconc, lrw,
     &          ipar
        double precision rwspec, rw, pl, pu, rpar
        dimension iwnconv(nnconv+1, np+1), iwspec(nspec+1, np+2), rw(lrw),
     &            iwconc(nconc+1, 2), rwspec(nspec+1), pl(np), pu(np)
        dimension rpar(*), ipar(*)

c       Bounds
```

```
      integer i, j

c     Lower bounds
      pl(1) = 0d0
      pl(2) = 0d0
      pl(3) = 0d0
      pl(4) = 0d0
      pl(5) = 1d-5
      pl(6) = 1d-5

c     Upper bounds
      pu(1) = 1d0
      pu(2) = 1d0
      pu(3) = 1d0
      pu(4) = 1d0
      pu(5) = 16d0
      pu(6) = 16d0

c     Initial point for local optimisation
      do i=1,np
        rw(i) = (pu(i)+pl(i))/2d0
      enddo

c     Problem parameters
      rpar(1) = 0.09755988d0
      rpar(2) = 0.99d0*rpar(1)
      rpar(3) = 0.0391908d0
      rpar(4) = 0.9d0*rpar(3)

c     Initialise working arrays
      do i=1,nnconv+1
        do j=1,np+1
          iwnconv(i,j) = 0
        enddo
      enddo
      do i=1,nspec+1
        do j=1,np+1
          iwspec(i,j) = 0
        enddo
        rwspec(i) = 0d0
      enddo
      do i=1,nconc+1
        do j=1,2
          iwconc(i,j) = 0
        enddo
      enddo

c           Define bilinear terms
```

```
c          # Bilinear term #1 : k1*x1*x5
             iwspec(1,1) = 1
             iwspec(1,2) = 1
             iwspec(1,6) = 1
             iwspec(1,8) = 1
             rwspec(1) = rpar(1)

c          # Bilinear term #2 : k2*x2*x6
             iwspec(2,1) = 3
             iwspec(2,3) = 1
             iwspec(2,7) = 1
             iwspec(2,8) = 2
             rwspec(2) = rpar(2)

c          # Bilinear term #3 : k3*x3*x5
             iwspec(3,1) = 5
             iwspec(3,4) = 1
             iwspec(3,6) = 1
             iwspec(3,8) = 3
             rwspec(3) = rpar(3)

c          # Bilinear term #4 : k4*x4*x6
             iwspec(4,1) = 7
             iwspec(4,5) = 1
             iwspec(4,7) = 1
             iwspec(4,8) = 4
             rwspec(4) = rpar(4)

c          Define univariate concave terms
c          # Concave term #1
             iwconc(1,1) = 9
             iwconc(1,2) = 5

c          # Concave term #2
             iwconc(2,1) = 9
             iwconc(2,2) = 6

         end
```

3.2 User-supplied subroutines

Several FORTRAN subroutines must be provided by the user in order to fully define the problem. The name of all user-defined subroutines can be changed at the user's will, as they are passed as arguments to `NLPGLOB`.

3.2.1 Subroutine `func`

The subroutine `func`, to be provided by the user, computes the value of the objective function and the constraints of the original NLP. The specification of subroutine `func` for `NLPGLOB` is as follow:

```
      subroutine func(np, nctr, p, f, g, rpar, ipar)

        implicit none
        integer np, nctr, ipar
        double precision p, f, g, rpar
        dimension p(np), g(nctr)
        dimension rpar(*), ipar(*)
c
c       Assign to f and g the value of the objective function
c       and constraints evaluated at p
c
        return
      end
```

Arguments:

`np` **[input]** Number of parameters.

`nctr` **[input]** Number of constraints.

`p(np)` **[input]** Parameter vector. When calling `func`, `p` contains the current iterate.

`f` **[output]** Objective function. The user has to evaluate the objective function at `p` and to store the result in `f`.

`g(nctr)` **[output]** Constraints. The user has to evaluate the constraints at `p` and to store the result in `g`. The constraints must be ordered as follow: first the `nctre` (possibly zero) **linear** equality constraints or nonlinear equality constraints that can be linearly underestimated (*i.e.* $g(\boldsymbol{p}) = 0$); then the `nctr-nctre` (possibly zero) nonlinear **negative** inequality constraints (*i.e.* $g(\boldsymbol{p}) \leq 0$). Note that nonlinear equality constraints including univariate concave or general nonconvex terms, if any, should be split as two nonlinear inequality constraints of opposite signs (*i.e.* $g(\boldsymbol{p}) \leq 0$ and $-g(\boldsymbol{p}) \leq 0$).

`rpar(*)` **[input/output]** User-defined real parameter vector that is not altered by `NLPGLOB` and can be used in communication among the user-supplied subroutines.

`ipar(*)` **[input/output]** User-defined integer parameter vector that is not altered by `NLPGLOB` and can be used in communication among the user-supplied subroutines.

Illustrating example (Example 1 continued).

For the problem defined by the equations (NLP1), the user-defined subroutine `func` is as follow.

```
      subroutine func(np, nctr, p, f, g, rpar, ipar)

        implicit none

c       Arguments
        integer np, nctr, ipar
        double precision p, f, g, rpar
        dimension p(np), g(nctr)
        dimension rpar(*), ipar(*)

c       Calculation of the values of the objective function and
c       constraints of the original NLP

        f = -p(4)
        g(1) = p(1)-1d0+rpar(1)*p(1)*p(5)
        g(2) = p(2)-p(1)+rpar(2)*p(2)*p(6)
        g(3) = p(3)+p(1)-1d0+rpar(3)*p(3)*p(5)
        g(4) = p(4)-p(3)+p(2)-p(1)+rpar(4)*p(4)*p(6)
        g(5) = (p(5)**0.5d0+p(6)**0.5d0)-4d0

      end
```

3.2.2 Subroutine grad

The subroutine `grad` computes the gradients of the objective function and the constraints of the original NLP with respect to the p variables. Also note that `x`, `f` and `g` are not allowed to be altered within `grad`. The specification of subroutine `grad` for `NLPGLOB` is as follow:

```
      subroutine grad(np, nctr, p, f, g, df, dg, rpar, ipar)

        implicit none
        integer np, nctr, ipar
        double precision p, f, g, df, dg, rpar
        dimension p(np), g(nctr), df(np), dg(nctr+1, np)
        dimension rpar(*), ipar(*)
c
c       Assign to df and dg the gradient of the objective function
c       and constraints evaluated at p
c
        return
      end
```

Arguments:

`np` **[input]** Number of parameters.

`nctr` **[input]** Number of constraints.

`p(np)` **[input]** Parameter vector. When calling `grad`, `p` contains the current iterate.

`f` **[input]** Objective function. When calling `grad`, `f` contains the actual objective function value.

`g(nctr)` **[input]** Constraints. When calling `grad`, `g` contains the actual constraint values.

`df(np)` **[output]** Gradient of the objective function. The user has to evaluate the gradient of the objective function at `p` and to store the result in `df`.

`dg(nctr+1,np)` **[output]** Gradient of the constraints. The user has to evaluate the gradient of the constraints at `p` and to store the result in `dg`. The constraints must ordered as in subroutine `func`.

`rpar(*)` **[input/output]** User-defined real parameter vector that is not altered by `NLPGLOB` and can be used in communication among the user-supplied subroutines.

`ipar(*)` **[input/output]** User-defined integer parameter vector that is not altered by `NLPGLOB` and can be used in communication among the user-supplied subroutines.

Note that whenever `grad` is called, a previous call of `func` was made at the same iterate `p` and `f`, `g` contain function values of the objective function and constraints at `p`.

Illustrating example (Example 1 continued).

For the problem defined by the equations (NLP1), the user-defined subroutine `grad` is as follow.

```
      subroutine grad(np, nctr, p, f, g, df, dg, rpar, ipar)

        implicit none

c       Arguments
        integer np, nctr, ipar
        double precision p, f, g, df, dg, rpar
        dimension p(np), g(nctr+1), df(np), dg(nctr+1, np)
        dimension rpar(*),ipar(*)

c       Counters
        integer i

c       Calculation of the gradients of the objective function and
c       constraints of the original NLP

        df(4) = -1d0

        dg(1,1) = 1d0+rpar(1)*p(5)
        dg(1,5) = rpar(1)*p(1)
```

```
          dg(2,1) = -1d0
          dg(2,2) = 1d0+rpar(2)*p(6)
          dg(2,6) = rpar(2)*p(2)

          dg(3,1) = 1d0
          dg(3,3) = 1d0+rpar(3)*p(5)
          dg(3,5) = rpar(3)*p(3)

          dg(4,1) = -1d0
          dg(4,2) = 1d0
          dg(4,3) = -1d0
          dg(4,4) = 1d0+rpar(4)*p(6)
          dg(4,6) = rpar(4)*p(4)

          dg(5,5) = 0.5d0*p(5)**(-0.5d0)
          dg(5,6) = 0.5d0*p(6)**(-0.5d0)

     end
```

3.2.3 Subroutine funcrel

The subroutine `funcrel` computes the value of the objective function and the constraints of the convex underestimated NLP. The specification of subroutine `funcrel` for `NLPGLOB` is:

```
        subroutine funcrel(np, nnconv, nps, nctr, nctrs, p, f, g,
       &                  pl, pu,alpha, rpar, ipar)

          implicit none
          integer np, nnconv, nspec, nctr, nctrs, ipar
          double precision p, f, g, alpha, rpar
          dimension p(np+nps), g(nctr+nctrs), alpha(np, nnconv+1)
          dimension pl(np), pu(np)
          dimension rpar(*), ipar(*)
c
c         Assign to f and g the value of the convex underestimated
c         objective function and constraints evaluated at p
c
          return
        end
```

Arguments:

`np` **[input]** Number of parameters.

`nnconv` **[input]** Number of general twice-differentiable nonconvex terms.

`nps` **[input]** Number of additional parameters due to the convex underestimation of bilinear, trilinear, fractional or trilinear fractional terms.

`nctr` [**input**] Number of constraints.

`nctrs` [**input**] Number of additional constraints due to the convex underestimation of bilinear, trilinear, fractional or trilinear fractional terms.

`p(np+nps)` [**input**] Parameter vector. The `nps` additional parameters are placed after the `np` parameters of the original NLP, and appears in the same order as defined in array `iwspec` (see § 3.1). When calling `funcrel`, `p` contains the current iterate.

`f` [**output**] Convex underestimated objective function. The user has to evaluate the convex underestimated objective function at `p` and to store the result in `f`.

`g(nctr+nctrs)` [**output**] Convex underestimated constraints. The user has to evaluate the convex underestimated constraints at `p` and to store the result in `g`. The ordrer of the constraints must be the same as in subroutine `func`: first the `nctre` (possibly zero) **linear** equality constraints; then the `nctr+nctrs-nctre` (possibly zero) linear or nonlinear **negative** inequality constraints.

`pl(np)` [**input**] Lower bounds on the parameters. When calling `funcrel`, `pl` contains the current lower bounds on the parameters `p`.

`pu(np)` [**input**] Upper bounds on the parameters. When calling `funcrel`, `pl` contains the current upper bounds on the parameters `p`.

`alpha(np,nnconv+1)` Shift parameters. When calling `funcrel`, `alpha` contains the actual values of the shift parameters α_j ($j = 1 \ldots n_{nconv}$) evaluated at `pl` and `pu` for each nonconvex terms and in the same order as defined in working array `iwnconv` (see § 3.1) and subroutine `inthess` (see § 3.2.5).

`rpar(*)` [**input/output**] User-defined real parameter vector that is not altered by `NLPGLOB` and can be used in communication among the user-supplied subroutines.

`ipar(*)` [**input/output**] User-defined integer parameter vector that is not altered by `NLPGLOB` and can be used in communication among the user-supplied subroutines.

Illustrating example (Example 1 continued).

For the convex underestimated problem defined by the equations (LBP1), *i.e.* by considering each nonconvex term of special structure (**Case 1**), the user-defined subroutine `funcrel` is as follow.

```
      subroutine funcrel(np, nnconv, nps, nctr, nctrs, p, f, g,
     &                   pl, pu, alpha, rpar, ipar)

        implicit none

c       Arguments
        integer np, nnconv, nps, nctr, nctrs, ipar
        double precision p, f, g, alpha, pl, pu, rpar
```

```
      dimension p(np+nps), g(nctr+nctrs+1), alpha(np, nnconv+1),
     &          pl(np+nps), pu(np+nps)
      dimension rpar(*), ipar(*)

c     Calculation of the values of the objective function and
c     constraints of the convex underestimated problem

      f = -p(4)
      g(1) = p(1)-1d0+rpar(1)*p(7)
      g(2) = p(2)-p(1)+rpar(2)*p(8)
      g(3) = p(3)+p(1)-1d0+rpar(3)*p(9)
      g(4) = p(4)-p(3)+p(2)-p(1)+rpar(4)*p(10)
      g(5) = (pl(5)**0.5d0+(pu(5)**0.5d0-pl(5)**0.5d0)
     &            /(pu(5)-pl(5))*(p(5)-pl(5))
     &        +pl(6)**0.5d0+(pu(6)**0.5d0-pl(6)**0.5d0)
     &            /(pu(6)-pl(6))*(p(6)-pl(6)))
     &        -4d0

c     Additional constraint set #1 : bilinear term p1*p5
      g(6) = (pu(1)*p(5)+pu(5)*p(1)-pu(1)*pu(5))-p(7)
      g(7) = (pl(1)*p(5)+pl(5)*p(1)-pl(1)*pl(5))-p(7)
      g(8) = p(7)-(pu(1)*p(5)+pl(5)*p(1)-pu(1)*pl(5))
      g(9) = p(7)-(pl(1)*p(5)+pu(5)*p(1)-pl(1)*pu(5))

c     Additional constraint set #2 : bilinear term p2*p6
      g(10) = (pu(2)*p(6)+pu(6)*p(2)-pu(2)*pu(6))-p(8)
      g(11) = (pl(2)*p(6)+pl(6)*p(2)-pl(2)*pl(6))-p(8)
      g(12) = p(8)-(pu(2)*p(6)+pl(6)*p(2)-pu(2)*pl(6))
      g(13) = p(8)-(pl(2)*p(6)+pu(6)*p(2)-pl(2)*pu(6))

c     Additional constraint set #3 : bilinear term p3*p5
      g(14) = (pu(3)*p(5)+pu(5)*p(3)-pu(3)*pu(5))-p(9)
      g(15) = (pl(3)*p(5)+pl(5)*p(3)-pl(3)*pl(5))-p(9)
      g(16) = p(9)-(pu(3)*p(5)+pl(5)*p(3)-pu(3)*pl(5))
      g(17) = p(9)-(pl(3)*p(5)+pu(5)*p(3)-pl(3)*pu(5))

c     Additional constraint set #4 : bilinear term p4*p6
      g(18) = (pu(4)*p(6)+pu(6)*p(4)-pu(4)*pu(6))-p(10)
      g(19) = (pl(4)*p(6)+pl(6)*p(4)-pl(4)*pl(6))-p(10)
      g(20) = p(10)-(pu(4)*p(6)+pl(6)*p(4)-pu(4)*pl(6))
      g(21) = p(10)-(pl(4)*p(6)+pu(6)*p(4)-pl(4)*pu(6))

     end
```

3.2.4 Subroutine `gradrel`

The subroutine `gradrel` computes the gradients of the objective function and the constraints of the convex underestimated NLP with respect to the p and ω variables. The specification of subroutine `gradrel` for `NLPGLOB` is as follow:

```
      subroutine gradrel(np, nnconv, nps, nctr, nctrs, p, f, g,
     &                   df, dg, pl, pu, alpha, rpar, ipar)

        implicit none
        integer np, nnconv, nps, nctr, nctrs, ipar
        double precision p, f, g, df, dg, pl, pu, alpha, rpar
        dimension p(np+nps), g(nctr+nctrs), df(np+nps),
     &            dg(nctr+nctrs+2, np+nps),
        dimension pl(np), pu(np), alpha(np, nnconv+1)
        dimension rpar(*), ipar(*)
c
c       Assign to df and dg the gradient of the convex underestimated
c       objective function and constraints evaluated at p
c
        return
      end
```

ARGUMENTS:

`np` [**input**] Number of parameters.

`nnconv` [**input**] Number of general twice-differentiable nonconvex terms.

`nps` [**input**] Number of additional parameters due to the convex underestimation of bilinear, trilinear, fractional or trilinear fractional terms.

`nctr` [**input**] Number of constraints.

`nctrs` [**input**] Number of additional constraints due to the convex underestimation of bilinear, trilinear, fractional or trilinear fractional terms.

`p(np+nps)` [**input**] Parameter vector. The `nps` additional parameters are placed after the `np` parameters of the original NLP, and appears in the same order as defined in array `iwspec` (see § 3.1). When calling `gradrel`, `p` contains the current iterate.

`f` [**input**] Convex underestimated objective function. When calling `gradrel`, `f` contains the actual convex underestimated objective function value.

`g(nctr+nctrs)` [**input**] Convex underestimated constraints. When calling `gradrel`, `g` contains the actual convex underestimated constraint values.

`df(np+nps)` [**output**] Gradient of the convex underestimated objective function. The user has to evaluate the gradient of the convex underestimated objective function at `p` and to store the result in `df`.

`dg(nctr+nctrs+2,np+nps)` **[output]** Gradient of the convex underestimated constraints. The user has to evaluate the gradient of the convex underestimated constraints at `p` and to store the result in `dg`. The constraints must be ordered as in subroutine `funcrel`.

`pl(np)` **[input]** Lower bounds on the parameters. When calling `gradrel`, `pl` contains the current lower bounds on the parameters `p`.

`pu(np)` **[input]** Upper bounds on the parameters. When calling `gradrel`, `pl` contains the current upper bounds on the parameters `p`.

`alpha(np,nnconv+1)` Shift parameters. When calling `gradrel`, `alpha` contains the actual values of the shift parameters α_j ($j = 1 \ldots n_{nconv}$) evaluated at `pl` and `pu` for each nonconvex terms and in the same order as defined in working array `iwnconv` (see § 3.1) and subroutine `inthess` (see § 3.2.5).

`rpar(*)` **[input/output]** User-defined real parameter vector that is not altered by `NLPGLOB` and can be used in communication among the user-supplied subroutines.

`ipar(*)` **[input/output]** User-defined integer parameter vector that is not altered by `NLPGLOB` and can be used in communication among the user-supplied subroutines.

Illustrating example (Example 1 continued).

For the convex underestimated problem defined by the equations (LBP1), *i.e.* by considering each nonconvex term of special structure (**Case 1**), the user-defined subroutine `gradrel` is as follow.

```
      subroutine gradrel(np, nnconv, nps, nctr, nctrs, p, f, g, df, dg,
     &                   pl, pu, alpha, rpar, ipar)

        implicit none

c       Arguments
        integer np, nnconv, nps, nctr, nctrs, ipar
        double precision p, f, g, df, dg, alpha, pl, pu, rpar
        dimension p(np+nps), g(nctr+nctrs+1), df(np+nps),
     &            dg(nctr+nctrs+2, np+nps), alpha(np, nnconv+1),
     &            pl(np+nps), pu(np+nps)
        dimension rpar(*), ipar(*)

c       Counters
        integer i

c       Calculation of the gradients of the objective function and
c       the constraints of the convex underestimated problem

        df(4) = -1d0
```

```
      dg(1,1) = 1d0
      dg(1,7) = rpar(1)

      dg(2,1) = -1d0
      dg(2,2) = 1d0
      dg(2,8) = rpar(2)

      dg(3,1) = 1d0
      dg(3,3) = 1d0
      dg(3,9) = rpar(3)

      dg(4,1) = -1d0
      dg(4,2) = 1d0
      dg(4,3) = -1d0
      dg(4,4) = 1d0
      dg(4,10) = rpar(4)

      dg(5,5) = (pu(5)**0.5d0-pl(5)**0.5d0)/(pu(5)-pl(5))
      dg(5,6) = (pu(6)**0.5d0-pl(6)**0.5d0)/(pu(6)-pl(6))

c     Additional constraint set #1 : bilinear term p1*p5
      dg(6,1) = pu(5)
      dg(6,5) = pu(1)
      dg(6,7) = -1d0

      dg(7,1) = pl(5)
      dg(7,5) = pl(1)
      dg(7,7) = -1d0

      dg(8,1) = -pl(5)
      dg(8,5) = -pu(1)
      dg(8,7) = 1d0

      dg(9,1) = -pu(5)
      dg(9,5) = -pl(1)
      dg(9,7) = 1d0

c     Additional constraint set #2 : bilinear term p2*p6
      dg(10,2) = pu(6)
      dg(10,6) = pu(2)
      dg(10,8) = -1d0

      dg(11,2) = pl(6)
      dg(11,6) = pl(2)
      dg(11,8) = -1d0

      dg(12,2) = -pl(6)
      dg(12,6) = -pu(2)
```

```
          dg(12,8) = 1d0

          dg(13,2) = -pu(6)
          dg(13,6) = -pl(2)
          dg(13,8) = 1d0

c         Additional constraint set #3 : bilinear term p3*p5
          dg(14,3) = pu(5)
          dg(14,5) = pu(3)
          dg(14,9) = -1d0

          dg(15,3) = pl(5)
          dg(15,5) = pl(3)
          dg(15,9) = -1d0

          dg(16,3) = -pl(5)
          dg(16,5) = -pu(3)
          dg(16,9) = 1d0

          dg(17,3) = -pu(5)
          dg(17,5) = -pl(3)
          dg(17,9) = 1d0

c         Additional constraint set #4 : bilinear term p4*p6
          dg(18,4) = pu(6)
          dg(18,6) = pu(4)
          dg(18,10) = -1d0

          dg(19,4) = pl(6)
          dg(19,6) = pl(4)
          dg(19,10) = -1d0

          dg(20,4) = -pl(6)
          dg(20,6) = -pu(4)
          dg(20,10) = 1d0

          dg(21,4) = -pu(6)
          dg(21,6) = -pl(4)
          dg(21,10) = 1d0

        end
```

3.2.5 Subroutine `inthess`

The subroutine `inthess` computes bounds on the elements of the Hessian matrices $\frac{\partial^2 NC_i}{\partial \boldsymbol{p}^2}$ of each twice-differentiable nonconvex terms of general structure NC_i $(i = 1 \dots n_{nconv})$ for parameters $\boldsymbol{p}$ varying in a given range $\left[\boldsymbol{p}^L, \boldsymbol{p}^U\right]$. The specification of `inthess` for `NLPGLOB` is as follow:

```
      subroutine inthess(inconv, np, pl, pu, hint, rpar, ipar)

        implicit none
        integer inconv, np, ipar
        double precision pl, pu, hint, rpar
        dimension pl(np), pu(np), hint(np, np, 2)
        dimension rpar(*), ipar(*)
c
c       Assign to hint lower and upper bound values on each element of
c       the Hessian matrix of the inconv-th general nonconvex term for
c       parameters varying in the range [pl,pu]
c
        return
      end
```

ARGUMENTS:

`inconv` **[input]** Index of the general twice-differentiable nonconvex term for which the interval Hessian matrix is to be computed.

`np` **[input]** Number of parameters.

`pl(np)` **[input]** Lower bounds on the parameters. When calling `inthess`, `pl` contains the current lower bounds on the parameters `p`.

`pu(np)` **[input]** Upper bounds on the parameters. When calling `inthess`, `pl` contains the current upper bounds on the parameters `p`.

`hint(np,np,2)` **[output]** Interval Hessian matrix. The user has to evaluate lower and upper bounds on each element of the Hessian matrix of the `inconv`-th nonconvex term (in the same order as defined in working array `iwnconv` and subroutines `funcrel` and `gradrel`) and to store the result in `hint`.

`rpar(*)` **[input/output]** User-defined real parameter vector that is not altered by `NLPGLOB` and can be used in communication among the user-supplied subroutines.

`ipar(*)` **[input/output]** User-defined integer parameter vector that is not altered by `NLPGLOB` and can be used in communication among the user-supplied subroutines.

Special attention is placed on the fact that the successful implementation of the algorithm requires the availability of a rigorous procedure for interval function evaluations. Therefore, the `NLPGLOB` package is connected to a FORTRAN77 interval arithmetic library, `INTLIB`, developed by Kearfott et al. [1994]. It is recommended to use the subroutines provided within `INTLIB` package, *e.g.* by carrying out interval calculation using the natural extension form of the functions. Further information about interval arithmetic computation can be

found, for example, in Ratschek and Rokne [1988] and Kearfott [1996].

ILLUSTRATING EXAMPLE (Example 1 continued).

The convex underestimated problem (LBP1) obtained by considering each term of special structure does not contain any general nonconvex term. Therefore, subroutine `inthess` can be defined as the following empty subroutine.

```
      subroutine hessint(inconv, np, pl, pu, hint, rpar, ipar)

        implicit none

c       Arguments
        integer np, inconv, ipar
        double precision pl, pu, hint, rpar
        dimension pl(np), pu(np), hint(np, np, 2)
        dimension rpar(*), ipar(*)

      end
```

3.2.6 Function `fconc`

The function `fconc` computes the values of the problem univariate concave terms UC_i ($i = 1 \dots n_{conc}$). The specification of function `fconc` for `NLPGLOB` is as follow:

```
      double precision function fconc(iconc, np, p, rpar, ipar)

        implicit none
        integer iconc, np, ipar
        double precision p, rpar
        dimension p(np)
        dimension rpar(*), ipar(*)
c
c       Assign to fconc the value of the iconc-th univariate concave
c       term evaluated at p
c
        return
      end
```

ARGUMENTS:

`iconc` [**input**] Index of the univariate concave term for which the value is to be computed.

`np` [**input**] Number of parameters.

`p(np)` [**input**] Parameter vector. When calling `fconc`, `p` contains the actual parameter values.

`rpar(*)` **[input/output]** User-defined real parameter vector that is not altered by `NLPGLOB` and can be used in communication among the user-supplied subroutines.

`ipar(*)` **[input/output]** User-defined integer parameter vector that is not altered by `NLPGLOB` and can be used in communication among the user-supplied subroutines.

ILLUSTRATING EXAMPLE (Example 1 continued).

For the problem defined by the equations (NLP1), the user-defined function `fconc` is as follow.

```
      double precision function fconc(iconc, np, p, rpar, ipar)

        implicit none

c       Arguments
        integer iconc, np, ipar
        double precision p, rpar
        dimension p(np)
        dimension ipar(*), rpar(*)

   c    Univariate concave term #1
        if (iconc.eq.1) then
          fconc = p(5)**0.5d0
          return

   c    Univariate concave term #2
        elseif (iconc.eq.2) then
          fconc = p(6)**0.5d0
          return
        endif

      end
```

3.2.7 Illustrating example (Example 1 continued)

For the problem (NLP1), the files have been compiled on a ATHLON 1.9 GHz P.C. with GNU/Linux operating system and the public domain NLP solver `SLSQP` is used.

A number of close optima exist for problem (NLP1) as shown in Tab. 3.1.

- If the univariate concave and bilinear terms are underestimated linearly, the αBB algorithm identifies the global minimum after 39 iterations and 0.78 CPU seconds, with a relative convergence tolerance of 10^{-5}, branching strategy 2 and bound update performed at each iteration. The corresponding Fortran code is given in file `EXAMPLES/Example1a.f` and results are given in file `EXAMPLES/Example1a.txt`.
  The progress of both lower and upper bounds during the iterative process is shown in Fig. 3.1 (left plot). Also shown in this figure is the rate of convergence of the algorithm (rigth plot).

Table 3.1: Global and local minima of reactor network design problem

	f	x_1	x_2	x_3	x_4	x_5	x_6
Global	-0.38881	0.772	0.517	0.204	0.389	3.036	5.097
Local 1	-0.38808	1.0	0.393	0.0	0.388	10^{-5}	15.975
Local 2	-0.37461	0.391	0.391	0.375	0.375	15.975	10^{-5}

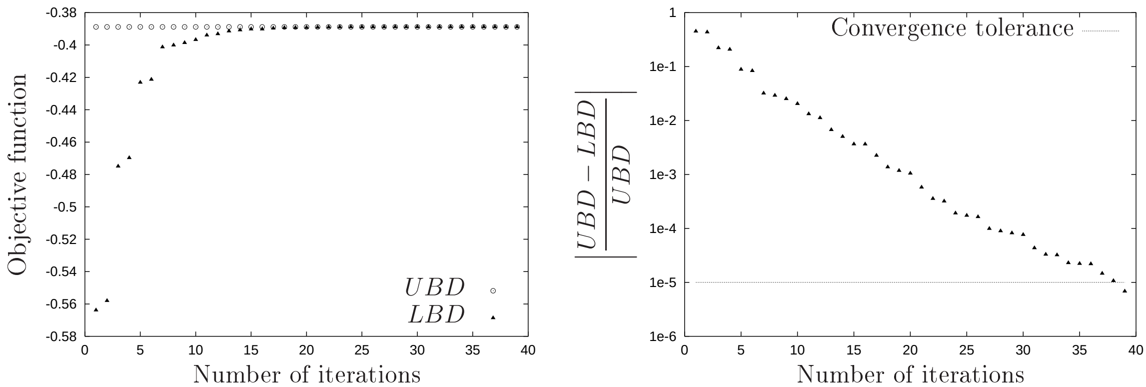

Figure 3.1: Progress of both upper and lower bounds for example (NLP1) (left plot) and rate of convergence of the αBB algorithm (right plot) when the univariate concave and bilinear terms are underestimated linearly.

- When bilinear terms are treated as nonconvex terms (univariate concave terms are still underestimated linearly), the computational requirements increase to 289 iterations and around 4.3 CPU seconds. The corresponding Fortran code is given in file `EXAMPLES/Example1b.f` and results are given in file `EXAMPLES/Example1b.txt`.

These results clearly demonstrate the need to construct the tight convex underestimators to reduce the computational requirements. Whenever possible, it is therefore greatly recommended to underestimate nonconvex terms of special structure linearly.

3.3 Organisation of files and subroutines

The user has to provide the principal routine that calls `NLPGLOB` as well as routines for system description (*e.g.* `main.f`). With these routines serving as the user information, the `NLPGLOB` package consists of several modules that can be combined together according to the needs of the particular problem. A possible project includes files `main.f`, `nlpglob.f`, `nlpsolv.f`, `calcalpha.f`, `branch.f`, `initaBB.f`, `slsqp.f`, `nlpql.f` (possibly empty) and `blaeis.f`, as well as files `d1i1mach.f`, `basicops.f`, `elemfuns.f`, `utilfuns.f` and `miscmach.f` from the `INTLIB` package.

3.3.1 Subroutines in `nlpglob.f`

The principal flow of information when the routine `nlpglob` is called is as follow:

- `NLPGLOB` (i) Initialises workspace, allocates space for the workspace vectors (`INIT001`); (ii) Depending on the value of `info(1)`, minimum workspace is calculated, gradient are checked (`CHKGRAD`), local optimisation is performed from a given starting point, or global optimisation is performed.
- `UNDEREST` Computes the α's for each general nonconvex term (see file `calcalpha.f`). Several methods can be implemented depending on the the value of `info(3)`.
- `BNDUPDT` Initialises the variable bound update procedure. The convex underestimated functions are updated for each variable (`UNDEREST`) and the variable bound update problem is solved (`NLPFEAS`) (see file `nlpsolv.f`). Variable bound update is performed at the onset of an αBB run or at each iteration depending on the value of `info(5)`.
- `BRANCHING` Initialises the branching procedure (see file `branch.f`). A reduced set of variable is selected according to the branching strategy, *i.e.* the value of `info(4)` (`BRANCH001`); the *least reduced axis* rule is applied (`BRANCH002`); the selected interval is then split into two sub-intervals (`BRANCH003`).
- `NODEUPDT` Updates the lower bound on the global optimum. The smallest lower bound is selected among all previously stored lower bounds.
- `CHKNODE` Checks whether the current upper bound relies within a feasible node of the branch-and-bound tree.

3.3.2 Subroutines in `nlpsolv.f`

The file `nlpsolv.f` contains subroutines to solve upper bounding, lower bounding and variable bound update problem locally. NLP solvers `SLSQP` (a public domain NLP solver) and `NLPQL` (that has to be bought directly from the author) are currently supported; the code remains unchanged in both cases. Note that any other NLP solver can be implemented provided that it supports a mechanism for reverse communication mode.

- `NLPLOC` Principal routine for the resolution of upper bounding problems. Calls the corresponding NLP solver via the value of `info(8)`, and requests the evaluation of the cost/constraint functions (`func`) and their gradients (`grad`, `gradfd`).
- `NLPREL` Principal routine for the resolution of lower bounding problems. Calls the corresponding NLP solver via the value of `info(8)`, and requests the evaluation of the convex underestimated cost/constraint functions (`funcrel`) and their gradients (`gradrel`, `gradfdr`).
- `NLPFEAS` Principal routine for the resolution of bound unpdate problems. Calls the corresponding NLP solver via the value of `info(8)`, and requests the evaluation of the convex underestimated cost/constraint functions (`funcrel`) and their gradients (`gradrel`, `gradfdr`).

`gradfd` Calculates gradient information by the method of centered finite differences for the original cost/constraint functions.

`gradfdr` Calculates gradient information by the method of centered finite differences for the convex underestimated cost/constraint functions.

3.3.3 Subroutines in `branch.f`

The file `branch.f` contains subroutines to perform the branching step of the branch-and-bound algorithm.

`BRANCH001` Determines the nonconvex term which appears to have the worst underestimator as a basis for the selection of the branching variables. Branching strategies 1,2,3 and 4 (see § 2.1) corresponds to different values of `info(4)`.

`BRANCH002` Determines the interval to be bisected through the *least reduced axis* rule (see § 2.1).

`BRANCH003` Performs bisection of a given axis into two sub-intervals; makes use of `INTLIB` routines.

`MIDPOINT` Computes the midpoint of a given interval.

`TYPESPEC` Determines the participating variables in each nonconvex term of special structure based on the information given in working array `iwspec`.

`SEPMAXNCONV` Computes the maximum separation distance between a given general nonconvex term and its convex underestimator.

`SEPOPTNCONV` Computes the separation distance between a given general nonconvex term and its convex underestimator at a given point.

`SEPMAXBL` Computes the maximum separation distance between a given bilinear term and its linear underestimator.

`SEPOPTBL` Computes the separation distance between a given bilinear term and its linear underestimator at a given point.

`SEPOPTTL` Computes the separation distance between a given trilinear term and its linear underestimator at a given point.

`SEPOPTFR` Computes the separation distance between a given bilinear fractional term and its linear underestimator at a given point.

`SEPOPTFRT` Computes the separation distance between a given trilinear fractional term and its linear underestimator at a given point.

`SEPMAXCONC` Computes the maximum separation distance between a given univariate concave term and its linear underestimator; makes use of user-defined function `fconc`.

`SEPOPTCONC` Computes the separation distance between a given univariate concave term and its linear underestimator at a given point; makes use of user-defined function `fconc`.

3.3.4 Subroutines in `calcalpha.f`

The file `calcalpha.f` computes the values of the α's for a given general twice-differentiable nonconvex term and a given parameter range based on its interval Hessian matrix.

`CALCALPHA` Principal routine for the computation of the α's of a given general nonconvex term. Several methods are currently implemented (see `info(3)`.

`GershUni` Implements the Gershgorin theorem to compute a lower bound on all the eigenvalues of the interval Hessian matrix of a given general nonconvex term.

`GershNun` Implements the scaled Gershgorin theorem to compute a lower bound on each eigenvalues of the interval Hessian matrix of a given general nonconvex term.

`MethEmat` Implements the E-matrix method to compute a lower bound on all the eigenvalues of the interval Hessian matrix of a given general nonconvex term.

`MethMorKok` Implements the Mori and Kokane's method to compute a lower bound on all the eigenvalues of the interval Hessian matrix of a given general nonconvex term.

`MethHessLB` Implements the lower bounding Hessian method to compute a lower bound on all the eigenvalues of the interval Hessian matrix of a given general nonconvex term.

`calcmatvert` Computes vertex matrices to apply Mori and Kokane's method.

`calcmatmod` Computes the modified radius and midpoint matrices of a given square interval matrix and adds the square matrix E.

`calcmatlb` Computes a lower bounding matrix of a given interval Hessian matrix.

`bornvp` Computes or orders the eigenvalues of a given symetric square matrix by using routine `drsp` (and related subroutines) from the eigensystem subroutine package (`eispack`) (see file `blaeis.f`).

`verdiag` Checks consistency of eigenvalues and eigenvectors computed by subroutine `drsp`.

3.3.5 Subroutines in `blaeis.f`

The code makes (heavy) use of `blas` and `eispack` routines (available from NETLIB). As the routines are de facto standard, they are usually installed locally in optimised version for the given processor and can be linked directly when creating executables. If it is not the case, the file `blaeis.f` contains all (unoptimised) needed routines and should be added to the project.

3.3.6 Reserved common blocks

The package does not introduce any common blocks. All information transfer is done by workspace vectors which organisation is specified in Appendix A. Each NLP solver introduces its own common blocks; the user is therefore asked to refer to their documentation for further details.

CHAPTER 4

Examples

Some examples will be presented here that can be used to check the `NLPGLOB` distribution, as well as to get acquainted with the desired file/subroutine specifications. Each example is associated with the code accompanying this manual. The corresponding programs files and results can be found in the `SOURCES/EXAMPLE/` directory.

All examples have been tested in GNU/Linux operating system with GNU g77 and Absoft Linux Compiler f77.

4.1 Example 2

The second example is detailed in order to illustrate the αBB algorithm. Problem (NLP2) consists of an unconstrained nonconvex optimisation problem with a global minimum and two local minima:

$$\begin{aligned} \min_p \; & \cos(14.5\,p - 0.3) + p\,(p + 0.2) \\ \text{s.t.} \; & -1 \;\leq\; p \;\leq\; 0 \end{aligned} \tag{NLP2}$$

The objective function is plotted in Fig. 4.1. Also shown on this figure are two convex underestimators of the objective function on subintervals $[-1\ \ -0.5]$ and $[-0.5\ 0]$ (with α parameters computed using the E-matrix method).

Computations are performed on Athlon 1.9MHz P.C., with NLP solver SLSQP and a tight relative convergence tolerance $\varepsilon = 10^{-6}$. When variable bound update is performed at each iteration, `NLPGLOB` identifies the global minimum of problem (NLP2) in 4 iterations only and 0.018 CPU seconds. These results are displayed in Fig. 4.2.

The corresponding Fortran program and computational results can be found in files `EXAMPLES/Example2.f` and `EXAMPLES/Example2.txt` respectively.

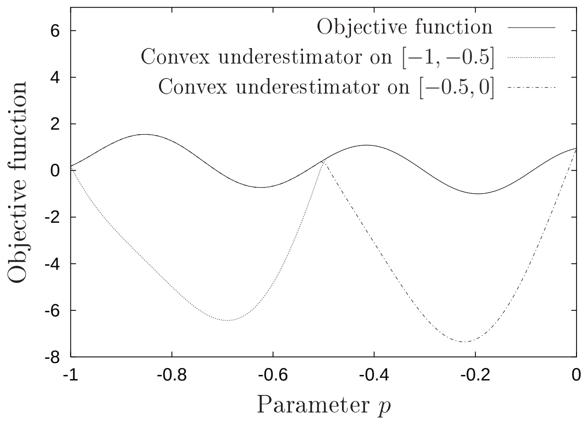

Figure 4.1: Objective function of problem (NLP2) and its convex underestimators over subintervals $[-1, -0.5]$ and $[-0.5, 0]$.

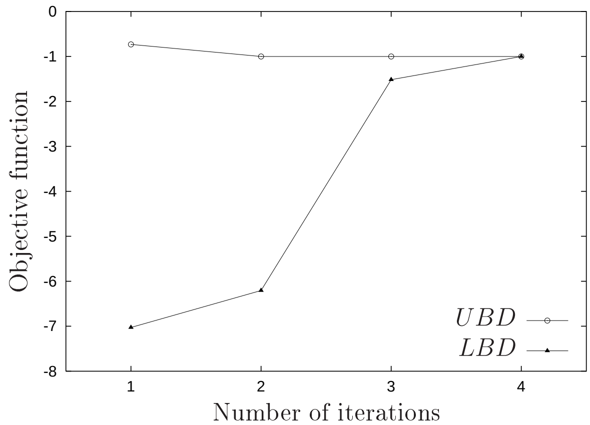

Figure 4.2: Progress of both upper and lower bounds for example (NLP2).

4.2 Example 3

The third illustrating example under consideration is the design of a three-stage process system with recycle, taken from [Ryoo and Sahinidis, 1995] and illustrated in Fig. 4.3

This problem can be formulated as a linearly constrained concave optimisation problem:

$$\begin{array}{ll} \min\limits_{\boldsymbol{p}} & p_1^{0.6} + p_2^{0.6} + p_3^{0.4} - 4p_3 + 2p_4 + 2p_5 - p_6 \\ \text{s.t.} & \begin{array}{rcl} -3p_1 + p_2 - 3p_4 & = & 0 \\ -2p_2 + p_3 - 2p_5 & = & 0 \\ 4p_4 - p_6 & = & 0 \\ p_1 + 2p_4 & \leq & 4 \\ p_2 + p_5 & \leq & 4 \\ p_3 + p_6 & \leq & 6 \\ (p_1, p_2, p_3, p_4, p_5, p_6) & \geq & (0,0,0,0,0,0) \\ (p_1, p_2, p_3, p_4, p_5, p_6) & \leq & (3,4,4,2,2,6) \end{array} \end{array} \tag{NLP3}$$

This global optimum of problem (NLP3) is obtained at $\boldsymbol{p} = [0.1666667, 2, 4, 0.5, 0, 2]^T$

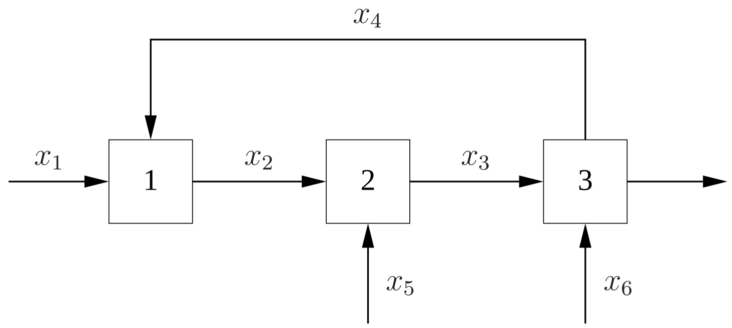

Figure 4.3: Design of three-stage process system with recycle.

with $f = -13.401904$.

Convex underestimation of nonconvex problem (NLP3) is obtained by simply underestimating linearly each concave term $p_1^{0.6}$, $p_2^{0.6}$ and $p_3^{0.4}$. The corresponding Fortran program and computational results can be found in files `EXAMPLES/Example3.f` and `EXAMPLES/Example3.txt` respectively.

This example illustrates the benefits that can be realised upon the application of variable bound update strategies. In every cases, computations are performed on ATHLON 1.9MHz P.C., with NLP solver SLSQP and a relative convergence tolerence $\varepsilon = 10^{-5}$.

- When no variable bound update is performed, `NLPGLOB` identifies the global optimum in 15 iterations and 0.019 CPU seconds.
- When variable bound update is performed at the onset of the `NLPGLOB` run, the global optimum is found within 4 iterations and the computational time is reduced to 0.010 CPU seconds.
- When variable bound update is performed at each iteration, only 2 iterations and 0.009 CPU seconds are required.

4.3 Example 4

The example of this section is taken from McDonald and Floudas [1994]. It correspond to a very challenging phase equilibrium problem, and is defined as follow: *Given i components participating in up to k potential phases under isothermal and isobaric conditions, find the mole vector n that minimises the value of the Gibbs energy while satisfying appropriate material balance constraints.*

The physical system under consideration describes the phase equilibrium of a system containing *n-Butyl-Acetate – Water*. The formulation is as follow:

$$
\begin{aligned}
\min_{\boldsymbol{n}} \ \hat{G} \ &= \ n_1^1 \ln n_1^1 + n_2^1 \ln n_2^1 - (n_1^1 + n_2^1) \ln (n_1^1 + n_2^1) \\
&+ n_1^2 \ln n_1^2 + n_2^2 \ln n_2^2 - (n_1^2 + n_2^2) \ln (n_1^2 + n_2^2) \\
&+ \mathcal{G}_{12}\tau_{12} n_1^1 \frac{n_2^1}{n_2^1 + \mathcal{G}_{12} n_1^1} + \mathcal{G}_{21}\tau_{21} n_2^1 \frac{n_1^1}{n_1^1 + \mathcal{G}_{21} n_2^1} \\
&+ \mathcal{G}_{12}\tau_{12} n_1^2 \frac{n_2^2}{n_2^2 + \mathcal{G}_{12} n_1^2} + \mathcal{G}_{21}\tau_{21} n_2^2 \frac{n_1^2}{n_1^2 + \mathcal{G}_{21} n_2^2} \\
\text{s.t.} \quad n_1^1 + n_1^2 \ &= \ 0 \\
n_2^1 + n_2^2 \ &= \ 0 \\
(n_1^1, n_1^2, n_2^1, n_2^2) \ &\geq \ (0, 0, 0, 0) \\
(n_1^1, n_1^2, n_2^1, n_2^2) \ &\leq \ (0.5, 0.5, 0.5, 0.5)
\end{aligned}
\tag{NLP4}
$$

where $\hat{G}(\boldsymbol{n}) = \frac{G(\boldsymbol{n})}{RT}$ (*i.e.* dimensionless G), n_i^k is the number of moles of species i present in phase k, τ_{ij} and $\mathcal{G}_{ij}$ are non-symmetric binary interaction parameters ; subscript $_1$ represents n-butyl-acetate while subscript $_2$ represents water, temperature is 298K and pressure is 1 atm; the initial mixture charge is equimolar ($n_i^T = 0.5 \ \forall i$) and no reaction occurs in the system ; the binary coefficients are taken from McDonald and Floudas [1994]:

$$
\begin{aligned}
\tau_{12} = 3.00498 \quad &, \quad \tau_{21} = 4.69071 \\
\mathcal{G}_{12} = 0.30794 \quad &, \quad \mathcal{G}_{21} = 0.15904
\end{aligned}
$$

The terms of the form $n_1 \ln n_1 + (n_1 + n_2) \ln (n_1 + n_2) = \ln \left(\frac{n_1}{n_1 + n_2} \right)$ can be shown to be convex, *e.g.* by using MAPLE[1]. On the other hand, the terms of the form $\frac{n_1 n_2}{n_1 + k n_2}$ can be shown to be concave. Therefore, the optimisation problem contains an objective function that has a convex term CT and 4 non-convex terms NT_i ($1 \leq i \leq 4$):

$$CT(\boldsymbol{n}) = n_1^1 \ln \frac{n_1^1}{n_1^1 + n_2^1} + n_2^1 \ln \frac{n_1^1}{n_1^1 + n_2^1} + n_1^2 \ln \frac{n_1^2}{n_1^2 + n_2^2} + n_2^2 \ln \frac{n_2^2}{n_1^2 + n_2^2} \tag{4.1a}$$

$$NT_1(\boldsymbol{n}) = \mathcal{G}_{12}\tau_{12} n_1^1 \frac{n_2^1}{n_2^1 + \mathcal{G}_{12} n_1^1} \tag{4.1b}$$

$$NT_2(\boldsymbol{n}) = \mathcal{G}_{12}\tau_{12} n_1^2 \frac{n_2^2}{n_2^2 + \mathcal{G}_{12} n_1^2} \tag{4.1c}$$

$$NT_3(\boldsymbol{n}) = \mathcal{G}_{21}\tau_{21} n_2^1 \frac{n_1^1}{n_1^1 + \mathcal{G}_{21} n_2^1} \tag{4.1d}$$

$$NT_3(\boldsymbol{n}) = \mathcal{G}_{21}\tau_{21} n_2^2 \frac{n_1^2}{n_1^2 + \mathcal{G}_{21} n_2^2} \tag{4.1e}$$

`NLPGLOB` identifies the global optimum $(n_1^1, n_1^2, n_2^1, n_2^2)^\star = (0.00071, 0.49928, 0.15586, 0.34414)$ in 965 iterations and 17.2 CPU seconds, with a relative convergence tolerance of 10^{-3} ; in this case, the α_i's are computed using the scaled Gerschgorin theorem, branching strategy 1 is used and variable bound updates are performed at each iteration. The corresponding Fortran main program and computational results can be found in files `EXAMPLES/Example4.f` and `EXAMPLES/Example4.txt` respectively.

---

[1] It can be checked that the function $x \ln \frac{x}{x+y}$ admits the following two eigenvalues $\lambda_1 = \frac{x^2+y^2}{xy(x+y)}$ and $\lambda_2 = 0$ and is therefore convex provided that $x \geq 0$ and $y \geq 0$.

The progress of both lower and upper bounds during the iterative process, as well as the rate of convergence of the algorithm, are shown in Fig. 4.4. The global optimum is found in 150 iterations. But although variable bound update is performed at each iteration, the progress of the lower bound is slow due to the strong nonconvexities of the terms NT_i for small values of the n_i^k's. In order to improve the convergence speed of the algorithm, problem (NLP4) can be reformulated by defining additional constraints so that general nonconvex terms are transformed into bilinear terms ; further details can be found in Maranas and Floudas [1994].

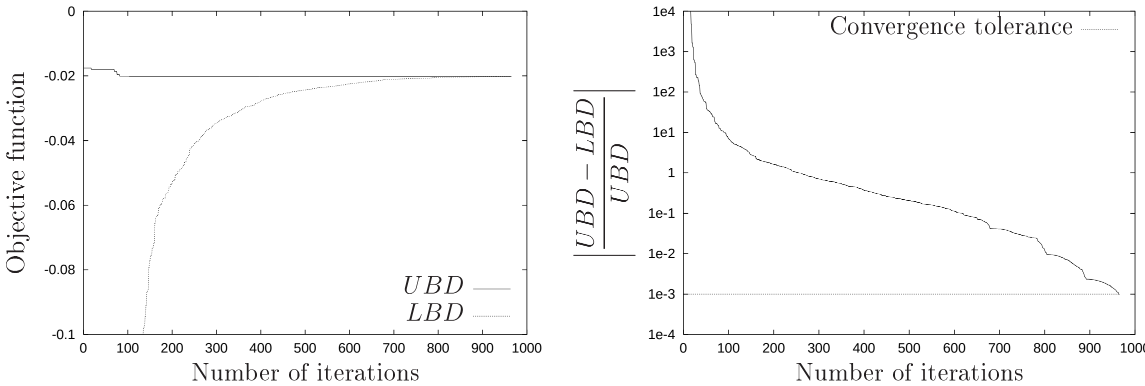

Figure 4.4: Progress of both upper and lower bounds for example (NLP4) (left plot) and rate of convergence of the αBB algorithm (right plot).

APPENDIX A

Workspace organisation

The following working arrays are used by `NLPGLOB`: `iw`(integer), `rw` (double precision) and `lw` (logical).

A.1 Positions in `iw`

Position	Lenght	Meaning
		αBB algorithm
`1`	`1`	pointer to the current best optimum
`2`	`1`	pointer to the lower bounds on the current best optimum
`3`	`1`	pointer to the upper bounds on the current best optimum
`4`	`1`	pointer to the solution of the current upper bounding problem
`5`	`1`	pointer to the lower bounds on the solution of the current upper bounding problem
`6`	`1`	pointer to the upper bounds on the solution of the current upper bounding problem

7	1	pointer to the shift parameter vector α of the 1st general nonconvex term
$\vdots$	$\vdots$	$\vdots$
`nnconv+7`	1	pointer to the shift parameters α_i of the `nnconv`th general nonconvex term
`nnconv+8`	1	pointer to the interval Hessian matrix of the 1st general nonconvex term
$\vdots$	$\vdots$	$\vdots$
`2*nnconv+8`	1	pointer to the interval Hessian matrix of the `nnconv`th general nonconvex term
`2*nnconv+9`	1	pointer to the vector used for the scaling of interval Hessian matrices the upper bounds on the solution of the current upper bounding problem
`2*nnconv+10`	1	pointer to the matrix used as the E-matrix for the calculation of the α's when applying the E-matrix method
`2*nnconv+11`	1	pointer to the workspace vector used to perform α calculation
$\vdots$	$\vdots$	$\vdots$
`2*nnconv+4*(2*k+r-2)+8`	1	pointer to the solution of the `r`th $\in \{1,2\}$ lower bounding problem at iteration `iter=k`$\in \{1,\ldots,$`maxit`$\}$
`2*nnconv+4*(2*k+r-2)+9`	1	pointer to the lower bounds on the solution of the `r`th $\in \{1,2\}$ lower bounding problem at iteration `iter=k`$\in \{1,\ldots,$`maxit`$\}$
`2*nnconv+4*(2*k+r-2)+10`	1	pointer to the upper bounds on the solution of the `r`th $\in \{1,2\}$ lower bounding problem at iteration `iter=k`$\in \{1,\ldots,$`maxit`$\}$
`2*nnconv+4*(2*k+r-2)+11`	1	pointer to the optimum value of the `r`th $\in \{1,2\}$ lower bounding problem at iteration `iter=k`$\in \{1,\ldots,$`maxit`$\}$
$\vdots$	$\vdots$	$\vdots$
		NLP solver
`2*nnconv+8*maxit+12`	1	pointer to the constraints at the optimum solution
`2*nnconv+8*maxit+13`	1	pointer to the gradient of the objective function at the optimum solution
`2*nnconv+8*maxit+14`	1	pointer to the gradient of the constraints at the optimum solution
`2*nnconv+8*maxit+15`	1	pointer to the values of the Lagrange multipliers at the optimum solution
`2*nnconv+8*maxit+16`	1	pointer to the approximation of the Hessian matrix of the Lagrange function at the optimum solution

`2*nnconv+8*maxit+17`	`1`	pointer to the diagonal of the approximation of the Hessian matrix of the Lagrange function at the optimum solution
`2*nnconv+8*maxit+18`	`1`	pointer to the workspace vector used to compute gradients using finite difference techniques
`2*nnconv+8*maxit+19`	`1`	pointer to the workspace vector used by the NLP solvers (SLSQP or NLPQL)
`2*nnconv+8*maxit+20`	`1`	pointer to the workspace vector used to perform gradient checks
`2*nnconv+8*maxit+21`	`liwnl`	integer workspace vector used by the NLP solvers (SLSQP or NLPQL)

integer workspace for NLP solvers:

```
liwnl = 2*np+2*nps+nctr+nctrs+20
```

A.2 Positions in `rw`

Position	Length	Meaning
		αBB algorithm
`iw(1)`	`np`	current best optimum
`iw(2)`	`np`	lower bounds on the current best optimum
`iw(3)`	`np`	upper bounds on the current best optimum
`iw(4)`	`np+nps`	solution of the current upper bounding problem
`iw(5)`	`np+nps`	lower bounds on the solution of the current upper bounding problem
`iw(6)`	`np+nps`	upper bounds on the solution of the current upper bounding problem
`iw(7)`	`np`	shift parameter vector α of the 1st general nonconvex term
$\vdots$	$\vdots$	$\vdots$
`iw(nnconv+7)`	`np`	shift parameter vector α of the `nnconv`th general nonconvex term
`iw(nnconv+8`	`2*np**2`	interval Hessian matrix of the 1st general nonconvex term
$\vdots$	$\vdots$	$\vdots$
`iw(2*nnconv+8)`	`2*np**2`	interval Hessian matrix of the `nnconv`th general nonconvex term
`iw(2*nnconv+9)`	`np`	vector used for the scaling of interval Hessian matrices the upper bounds on the solution of the current upper bounding problem

iw(2*nnconv+10)	np**2	matrix used as the E-matrix for the calculation of the α's when applying the E-matrix method
iw(2*nnconv+11)	lrwca	double precision workspace vector used to perform α calculation
$\vdots$	$\vdots$	$\vdots$
iw(2*nnconv+4*(2*k+r-2)+8)	np+nps	solution of the $\mathtt{r}^{\mathrm{th}} \in \{1,2\}$ lower bounding problem at iteration iter=k$\in \{1,\ldots,$maxit$\}$
iw(2*nnconv+4*(2*k+r-2)+9)	np+nps	lower bounds on the solution of the $\mathtt{r}^{\mathrm{th}} \in \{1,2\}$ lower bounding problem at iteration iter=k$\in \{1,\ldots,$maxit$\}$
iw(2*nnconv+4*(2*k+r-2)+10)	np+nps	upper bounds on the solution of the $\mathtt{r}^{\mathrm{th}} \in \{1,2\}$ lower bounding problem at iteration iter=k$\in \{1,\ldots,$maxit$\}$
iw(2*nnconv+4*(2*k+r-2)+11)	1	optimum value of the $\mathtt{r}^{\mathrm{th}} \in \{1,2\}$ lower bounding problem at iteration iter=k$\in \{1,\ldots,$maxit$\}$
$\vdots$	$\vdots$	$\vdots$
		NLP solver
iw(2*nnconv+8*maxit+12)	nctr+nctrs+2	constraints at the optimum solution
iw(2*nnconv+8*maxit+13)	np+nps+1	gradient of the objective function at the optimum solution
iw(2*nnconv+8*maxit+14)	(np+nps+1)*(nctr+nctrs+2)	gradient of the constraints at the optimum solution
iw(2*nnconv+8*maxit+15)	(nctr+nctrs+1)+2*(np+nps)+2	Lagrange multipliers at the optimum solution
iw(2*nnconv+8*maxit+16)	(np+nps+1)**2	approximation of the Hessian matrix of the Lagrange function at the optimum solution
iw(2*nnconv+8*maxit+17)	np+nps+1	diagonal of the approximation of the Hessian matrix of the Lagrange function at the optimum solution
iw(2*nnconv+8*maxit+18)	np+nps+nctr+nctrs+2	double precision workspace vector used to compute gradients using finite difference techniques
iw(2*nnconv+8*maxit+19)	lrwnl	double precision workspace vector used by the NLP solvers (SLSQP or NLPQL)
iw(2*nnconv+8*maxit+20)	(np+nps+1)*(nctr+nctrs+2)	double precision workspace vector used to perform gradient checks

double precision workspace for α calculation:

```
lrwca = 4*np**2+3*np+2+np*(np+1)/2+1
```

double precision workspace for NLP solvers:

```
      lrwnl1 = mmax*n+4*mmax+4*m+18*n+55
   &          +3*nmax*nmax/2+10*nmax+2*mmax
      lrwnl2 = (3*nmax+m)*(nmax+1)+(nmax-me+1)*(mine+2)+2*mine
   &          +(nmax+mine)*(nmax-me)+2*me+nmax+(n+1)*n/2+2*m+3*n
   &          +3*nmax+1
      lrwnl = max(lwnl1,lwnl2)
```

A.3 Positions in `lw`

Position	Length	Meaning
`1`	`np`	logical workspace vector used for branching
`np+1`	`llwnl`	logical workspace vector used by the NLP solvers (SLSQP or NLPQL)
`np+2*nctr+2*nctrs+20`	`2*maxit`	logical workspace vector used to store the potential nodes of the branch-and-bound tree

logical workspace for NLP solvers:

```
      llwnl = 2*(nctr+nctrs+2)+15
```

BIBLIOGRAPHY

C.S. Adjiman, I.P. Androulakis, and C.A. Floudas. A global optimization method, αBB, for general twice-differentiable constrained NLPs – II. implementation and computational results. *Comput. & Chem. Engng.*, 22:1159–1179, 1998a.

C.S. Adjiman, S. Dallwig, C.A. Floudas, and Neumaier A. A global optimization method, αBB, for general twice-differentiable constrained NLPs – I. theoretical advances. *Comput. & Chem. Engng.*, 22:1137–1158, 1998b.

I.P. Androulakis, C.D. Maranas, and C.A. Floudas. αBB: A global optimization method for general constrained nonconvex problems. *J. Global Opt.*, 7(4), 1995.

R. Horst and H. Tuy. *Global Optimization: Deterministic Approaches*. Springer, New York, 3rd edition edition, 1996.

R.B. Kearfott. *Rigorous global search: Continuous problems*, volume 13 of *Nonconvex optimization and its applications*. Kluwer Academic Publishers, Dordrecht, Netherlands, 1996.

R.B. Kearfott, M. Dawande, K. Du, and Ch. Hu. `INTLIB`: a portable `FORTRAN77` elementary function library. *ACM Trans. Math. Software*, 20(4):447–459, 1994.

D. Kraft. A software package for Sequential Quadratic Programming. Technical Report DFVLR-FB 88-28, Deutsches Zentrum für Luft- und Raumfahrt, Köln, Germany, 1988.

C.D. Maranas and C.A. Floudas. Global minimum potential energy conformations of small molecules. *J. Global Opt.*, 4:135–170, 1994.

G.P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I – convex underestimating problems. *Mathematical Programming*, 10:147–175, 1976.

C.M. McDonald and C.A. Floudas. Decomposition based and branch-and-bound global optimization approaches for the phase equilibrium problem. *J. Global Opt.*, 5:205, 1994.

H. Ratschek and J. Rokne. *Computer methods for the range of functions.* Ellis Horwood Series in Mathematics and its Applications. Halsted Press, 1988.

H.S. Ryoo and N.V. Sahinidis. Global optimization of nonconvex NLPs and MINLPs with application in process design. *Comput. & Chem. Engng.*, 19(5):551–566, 1995.

K. Schittkowski. `NLPQL` : a `FORTRAN` subroutine solving constrained nonlinear programming problems. *Annals of Operations Research*, 5(6):485–500, 1985.

D. Ternet and L.T. Biegler. Interior point methods for reduced hessian successive quadratic programming. *Comput. & Chem. Engng.*, 23:859, 1999.