Construction of Convex Function Relaxations Using Automated Code Generation Techniques

Edward P. Gatzke, John E. Tolsma and Paul I. Barton

11th June 2001

Abstract

This paper describes how the automated code generation tool DAEPACK can be used to construct convex function relaxations of codes with embedded nonconvex functions. Modern deterministic global optimization algorithms involving continuous and/or integer variables often require such convex function relaxations. Within the described framework, the user supplies a code implementing the objective and constraints of a nonconvex optimization problem. DAEPACK then analyzes this code and automatically generates a collection of subroutines based upon various symbolic transformations used by automatic convexification algorithms. The convexification methods considered include the convex underestimators of McCormick, αBB of Floudas and coworkers, and the linearization strategy of Tawarmalani and Sahinidis. It should be noted that the user supplied code can be quite complex, including arbitrary nonlinear expressions, subroutines, and iterative loops.

The code generation approach has the advantage that it can be applied to general, legacy models coded in programming languages such as FORTRAN. It also provides

a generic symbolic transformation service for researchers interested in developing new global optimization algorithms. Numerical results are presented, including a study of how these techniques can be used to generate convex underestimators based on a hybridization of αBB and the method of McCormick.

KEYWORDS Global Optimization, Convex Relaxation, Automatic Code Generation, DAEPACK

1 Introduction

Global optimization problems arise in many applications related to process design, process operation, and computational chemistry. Deterministic methods for solving nonconvex optimization problems have progressed greatly over the past few decades. Branch-and-bound methods have been developed for nonconvex problems involving continuous variables and/or integer variables. Outer approximation methods have recently been extended to the solution of mixed-integer problems where the participating functions are nonconvex. Both the branch-and-bound and outer approximation methods require creation of convex function relaxations.

The seminal branch-and-bound approach [5] relies upon the generation of a convex lower bounding problem over a specified region. As the solution space of continuous and integer variables is partitioned by branching, the value of the convex underestimating functions must approach the value of the original nonconvex function. Local search methods can be used in the upper bounding problem to determine feasible solutions of the original nonconvex problem. The algorithm converges when the lower bound for all partitions is greater than (or within ϵ of) the current upper bound. The branch-and-reduce method [15] significantly improved the basic branch-and-bound method by developing methods to reduce the partition

sizes, making more nonconvex programming problems computationally tractable.

For mixed-integer nonlinear programming problems, the outer approximation decomposition method [3, 6] in some cases can be more efficient than branch-and-bound methods. This method has been limited to problems with convex constraints and convex objective function. Recently, the outer approximation method was extended to problems where the participating functions are nonconvex [10]. The mixed-integer nonconvex problem can be relaxed to create a convex lower bounding problem. As the algorithm proceeds, a sequence of upper bounding problems involving only continuous variables must be solved with the integer variables fixed. Some of these upper bounding problems will require solution of a global nonconvex problem only involving continuous values. Obviously, both the branch-and-bound and the outer approximation methods rely heavily upon creation of convex underestimating functions for nonconvex problems.

Three general purpose convexification methods are considered in this paper. A general method for convexification of factorable nonconvex functions has been developed [11, 16]. This method generates convex functions using the known convex envelopes of simple nonlinear functions. The original nonconvex problem is reformulated to a standard form with constraints involving simple nonlinear functions by the introduction of new variables and constraints. The αBB method [2] is another convexification approach that produces convex relaxations from general twice-differentiable nonconvex functions. This method does not add additional variables or constraints to the problem formulation for complex expressions, but it does require calculation of the minimum eigenvalue for the Hessian of the nonconvex function over the region of interest. Linearization of nonlinear functions has also been proposed as a convex underestimating technique [17]. This method takes advantage of the robustness and scalability of Linear Programming technology to develop a lower bound for the region

of interest.

A new hybrid relaxation approach is now proposed, developed from the combination of two relaxation techniques. In some cases, it is advantageous to generate relaxations using both the αBB approach and the McCormick method. The αBB relaxations can be extremely loose for constraints where the minimum eigenvalue of the Hessian is poorly estimated by interval analysis methods. McCormick based approaches can develop loose function relaxations in cases involving complex factorable nonlinear functions. Use of both types of convex relaxations will create redundant relaxations but can produce tighter relaxations than either method alone. Additionally, a tighter convex relaxation can be developed by application of the αBB method to complex intermediate nonlinear expressions that arise in the reformulation using the McCormick approach.

This paper also presents convexification extensions of the DAEPACK package, a component library originally developed to support simulation of legacy FORTRAN models. Many legacy models have been developed with extensive effort and meticulous care. These validated models often contain proprietary or classified information. DAEPACK provides automatic source-to-source transformation of models given source code. A variety of symbolic transformations are available, including automatic differentiation [8], interval analysis [12], and generation of convex function relaxations. This paper focuses on methods and application of convex relaxation using DAEPACK. Use of the FORTRAN language for model development supports a truly open general purpose modeling environment. DAEPACK generates automatically many of the components needed by developers of new global optimization techniques.

2 Relaxation Methods

McCormick [11] presents a method for generating convex relaxations of factorable composite functions of the form

$$T\left[t(\mathbf{x})\right] + U\left[u(\mathbf{x})\right] \cdot V\left[v(\mathbf{x})\right] \tag{1}$$

where $t(\mathbf{x})$, $u(\mathbf{x})$, and $v(\mathbf{x})$ are continuous scalar valued functions of $\mathbf{x} \in \mathbb{R}^n$. $T[\cdot]$, $U[\cdot]$, and $V[\cdot]$ are continuous scalar valued functions mapping $\mathbb{R} \to \mathbb{R}$. It is assumed that convex underestimating and concave overestimating functions of $\mathbf{x}$ are available to bound $t(\mathbf{x})$, $u(\mathbf{x})$, and $v(\mathbf{x})$. Constant valued upper and lower bounds for $t(\mathbf{x})$, $u(\mathbf{x})$, and $v(\mathbf{x})$ must also be available on the region $\mathbf{x} \in S$ with S convex. Under these assumptions, convex and concave bounds on the original composite function can be derived as a function of $\mathbf{x}$. As one reduces the bounds on $\mathbf{x}$, the convex and concave bounding functions will converge to the original function. As stated in [11], the convex expression derived from Equation 1 may not be continuously differentiable, although an equivalent form can be derived through addition of new variables and inequality constraints.

This method can be used for arbitrarily complex expressions by recursive application. In general, convex underestimating and concave overestimating functions are not known for complex functions. However, for simple nonlinear expressions, convex underestimating and concave overestimating functions are available for many classes of functions. The original nonlinear function can be expressed in simpler terms by repeated introduction of new variables and convex constraints. Recursive methods for reformulation are described in [16, 17], and will be referred to in this paper as the Basic Reformulation.

2.1 Basic Reformulation

Given a FORTRAN code implementing an arbitrary nonlinear factorable function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$, a tree representation can be developed via an automatic analysis as described in [13, 18], based upon the notion of elementary functions. In the current implementation, the operators for addition, subtraction, multiplication, division, and exponentiation are supported, as well as the intrinsic functions $\ln(x)$ and e^x. A recursive algorithm is used to develop an equivalent representation of $f(\mathbf{x})$ in terms of linear functions and simple nonlinear functions. This is accomplished by introducing new variables representing simple expressions at the extremities of the tree. The convex envelopes for the simple nonlinear functions are known, so that the convex relaxation of $f(\mathbf{x})$ can be expressed in terms of convex expressions.

Bounds can be developed for a new variable based upon the known bounds of the variables involved in the simple expression. Inequality constraint functions can also be created based on the convex and concave envelopes of the simple nonlinear expression, serving as convex and concave constraint functions on the new variable. These inequality constraints are dependent upon the variables involved in the simple expression.

2.1.1 Basic Example

Figure 1 shows a tree representation for a factorable nonlinear term in the form:

$$f(x_1, x_2, x_3) = \frac{x_1}{\left(x_2 x_3 \frac{(x_2+x_3)}{2}\right)^{1/3}}$$

This example is motivated by terms appearing in a heat exchanger network synthesis problem formulated by [20]. The recursive reformulation algorithm introduces new variables $\mathbf{w}$

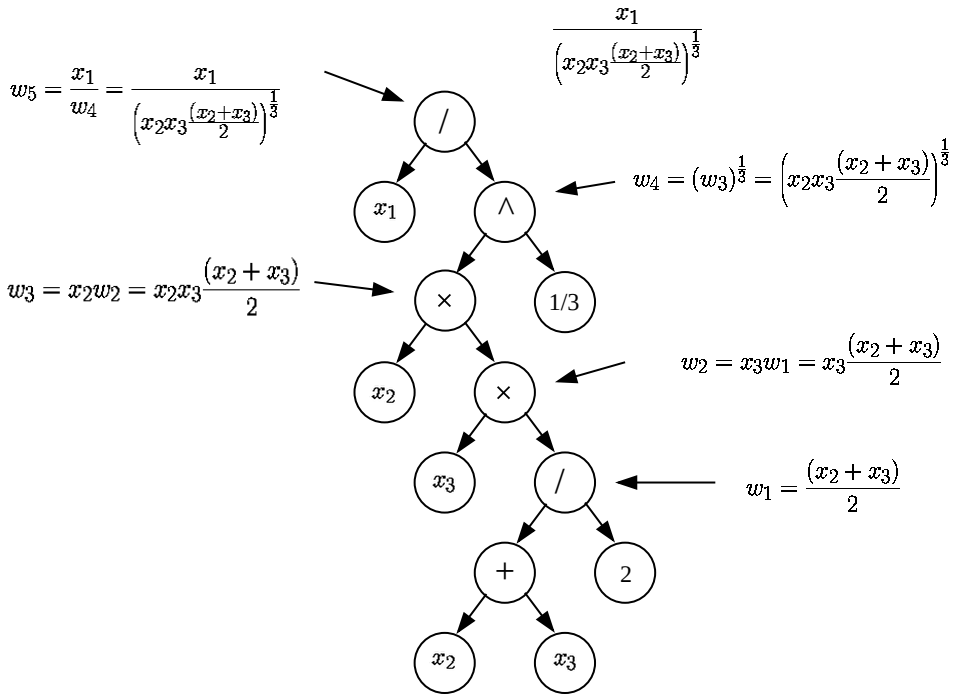

Figure 1: Equation tree representation for $x_1 \left(x_2 x_3 \frac{(x_2+x_3)}{2}\right)^{-1/3}$ including simple linear and nonlinear expressions for subtrees.

representing portions of the overall tree, or subtrees. Each subtree resulting from a nonlinear operation can be expressed in both simple terms involving one or two variables or in the original complex composite form involving only the original variables.

The new linear and nonlinear variables, $\mathbf{w} \in \mathbb{R}^l$, can augment the original variables $\mathbf{x}$ as $\mathbf{y} = \left[\mathbf{w}^T \, \mathbf{x}^T\right]^T \in \mathbb{R}^{l+n}$. For a nonlinear function $f(\mathbf{x})$ with $\{\mathbf{x} | \mathbf{x}^L \leq \mathbf{x} \leq \mathbf{x}^U\}$, the function in terms of linear and simple nonlinear constraints can be written as:

$$f(\mathbf{x}) = c^T \mathbf{y} \tag{2}$$

$$A\mathbf{y} = 0$$

$$\mathbf{y}^L \leq \mathbf{y} \leq \mathbf{y}^U$$

$$
\begin{aligned}
y_i &\equiv y_j\, y_k & \forall i,j,k \in \tau_{\text{bt}} \\
y_i &\equiv \tfrac{y_j}{y_k} & \forall i,j,k \in \tau_{\text{ft}} \\
y_i &\equiv y_j^{c_i} & \forall i,j \in \tau_{\text{pt}} \\
y_i &\equiv \text{fn}_i(y_j) & \forall i,j \in \tau_{\text{ut}}
\end{aligned}
$$

This transformed representation includes new linear constraints, bounds on the variables, and simple nonlinear equality constraints consisting of bilinear terms, fractional terms, power terms, and univariate terms. The convex and concave envelopes are known for these simple nonlinear expressions as functions of the variables and their upper and lower bounds. A convex relaxation of the original function can be realized by replacing the simple nonlinear equality constraints in Equations 2 with the convex underestimating functions $\check{g}_i$ and concave overestimating functions $\hat{g}_i$ for the nonlinear equality constraints as

$$
\begin{aligned}
\check{g}_i(y_j,y_k,y_j^L,y_j^U,y_k^L,y_k^U) &\le y_i \le \hat{g}_i(y_j,y_j,y_j^L,y_j^U,y_k^L,y_k^U) & \forall i,j,k \in \tau_{\text{bt}} \\
\check{g}_i(y_j,y_k,y_j^L,y_j^U,y_k^L,y_k^U) &\le y_i \le \hat{g}_i(y_j,y_k,y_j^L,y_j^U,y_k^L,y_k^U) & \forall i,j,k \in \tau_{\text{ft}} \\
\check{g}_i(y_j,y_j^L,y_j^U) &\le y_i \le \hat{g}_i(y_j,y_j^L,y_j^U) & \forall i,j \in \tau_{\text{pt}} \\
\check{g}_i(y_j,y_j^L,y_j^U) &\le y_i \le \hat{g}_i(y_j,y_j^L,y_j^U) & \forall i,j \in \tau_{\text{ut}}
\end{aligned}
\tag{3}
$$

Convex and concave relaxations for a variety of simple nonlinear terms are given in Appendix A. Interval bounds for variables involved in simple nonlinear expressions are also described in Appendix A.

This relaxation method is quite useful in that it can be used for any complex expression that can be represented as a finite sequence of instructions in a computer code involving a variety of standard operators and intrinsic functions with known convex and concave envelopes. Others have described this relaxation method applied to high level declarative modeling languages [16, 17], while this work extends such methods to general purpose procedural FORTRAN models. The Basic Reformulation does introduce new constraints and variables,

potentially increasing the problem size and computational complexity. Some functions, such as the gamma distribution, cannot be bounded using this method because they cannot be represented by the representation described above. Additionally, symbolic representations are not necessarily unique, as pointed out in [11]. The Basic Reformulation method may over-relax a function for a given representation. The function x^2 could be represented as $x \cdot x$, which would be mis-underestimated as a bilinear term rather than using x^2, the tightest convex underestimating function.

2.1.2 Subtree Expressions

Typically, a global optimization procedure would classify nonlinear constraints as convex or nonconvex, applying a reformulation method to only the nonconvex constraints. Nonlinear equality constraints are nonconvex. Nonlinear inequality constraints in the form $f(\mathbf{x}) \leq 0$ are convex if $f(\mathbf{x})$ is twice-continuously differentiable and the Hessian of $f(\mathbf{x})$ is positive semidefinite (nonnegative eigenvalues) for $\{\mathbf{x} | \mathbf{x}^L \leq \mathbf{x} \leq \mathbf{x}^U\}$.

It should be noted that the nonlinear equality expressions representing the function subtrees from Equations 2 may also be written only in terms of the original variables $\mathbf{x}$. In some cases, a subtree expression may be purely convex or concave over the original range of $\mathbf{x}$. In other cases, as an optimization procedure progresses, one or more subtree expressions may become purely convex or concave over the current range of $\mathbf{x}$. In these instances, the problem reformulation could potentially use the original subexpression as a upper or lower bound for y_i rather than relaxing the constraints on y_i using the underestimating function $\check{g}_i$ or the overestimating function $\hat{g}_i$. For the original range of $\mathbf{x}$, the Hessian eigenvalue bounds for each subtree expression resulting from the Basic Reformulation could be analyzed, potentially providing additional convex or concave constraints for some complex nonlinear terms

in the reformulation. Hessian information is currently under development for inclusion in future releases of DAEPACK which could automate this procedure.

In Example 1, the subtree expression for w_4 may be written as $w_4 = (w_3)^{\frac{1}{3}}$. This could also be expressed in terms of $\mathbf{x}$ as $w_4 = (x_2 x_3 (x_2 + x_3)/2)^{\frac{1}{3}}$. It can be verified that this expression is concave given the original bounds on x_2 and x_3, $10 \leq x_2 \, , x_3 \leq 280$. Therefore $(x_2 x_3 (x_2 + x_3)/2)^{\frac{1}{3}}$ is a valid concave upper bound for w_4, resulting in the convex constraint $w_4 - (x_2 x_3 (x_2 + x_3)/2)^{\frac{1}{3}} \leq 0$. Using the Basic Reformulation method, the upper bounding function for w_4 would depend upon w_3, which would be relaxed in the bilinear expressions for the w_3 and w_2 subtrees.

2.2 αBB Reformulation

The αBB based reformulation of Adjiman *et. al* [2] presents an alternative method for creation of convex and concave bounds for complex nonlinear expressions. This method does not require the introduction of new variables or constraints for complex expressions, but does require that the complex function be twice-continuously differentiable and lower bounds be calculated on the eigenvalues of the interval Hessian expressions for each complex nonlinear term.

The nonlinear function $f(\mathbf{x})$ can be written as a sum of linear terms, bilinear terms, linear fractional terms, simple univariate nonlinear terms, and complex nonlinear terms.

$$\begin{aligned} f(\mathbf{x}) = \quad & \mathbf{c}^T \mathbf{x} + \sum_{i=1}^{\mathrm{bt}} B_i x_{B1_i} x_{B2_i} + \sum_{i=1}^{\mathrm{ft}} F_i \frac{x_{F1_i}}{x_{F2_i}} \\ & + \sum_{i=1}^{\mathrm{ut}} f_{U_i}(x_{U_i}) + \sum_{i=1}^{\mathrm{nt}} f_{N_i}(\mathbf{x}) \end{aligned}$$

The complex nonlinear terms $f_{N_i}(\mathbf{x})$ are assumed to be twice-continuously differentiable. A

lower bounding function for $f_{N_i}(\mathbf{x})$ is given by

$$L(\mathbf{x}) \quad \leq \quad f_{N_i}(\mathbf{x}) + \sum_{j=1}^{n} \alpha_i \left(x_j^L - x_j\right)\left(x_j^U - x_j\right)$$

where α_i is chosen such that the Hessian of $L(\mathbf{x})$ is positive semidefinite. This can be accomplished by calculating a lower bound on the eigenvalues λ_j of the Hessian of complex function $f_{N_i}(\mathbf{x})$.

$$\alpha_i \geq \max\left\{0, -\frac{1}{2} \min_{\mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u} \lambda_j\right\}$$

Calculating the exact values of the eigenvalues of the Hessian over the range of $\mathbf{x}$ can be computationally demanding. As a result, a variety methods have been proposed in [1, 9] for deriving lower bounds for the eigenvalues of the Hessian function, given interval bounds on the elements of the Hessian matrix. While the αBB method does not require the introduction of numerous variables and constraints for complex nonlinear terms, calculating eigenvalue bounds for the interval Hessian adds computational complexity that must be repeated as the bounds for $\mathbf{x}$ change in a spatial branch-and-bound algorithm. If a given partition explored by the branch-and-bound algorithm results in a positive bound on the interval Hessian eigenvalues, the αBB method produces the tightest underestimating function for the expression as $\alpha_i = 0$.

2.3 Simple Hybrid Reformulation

The Basic Reformulation and αBB methods are both limited in that the function $f(\mathbf{x})$ may be relaxed more than is necessary (*i.e.* beyond its convex envelope) resulting in poor lower bounds for a given partition explored by a branch-and-bound algorithm. A poor lower bound for a partition forces increased partitioning, resulting in additional iterations.

It is desirable to provide the tightest possible bounds for a function using continuously differentiable functions. A hybrid approach is proposed here to mitigate this over relaxation of nonlinear terms.

In the Simple Hybrid Reformulation, both the Basic Reformulation and αBB Reformulation methods are applied simultaneously to complex nonlinear terms appearing in the function $f(\mathbf{x})$. As in the αBB Reformulation, bilinear, linear fractional, and simple univariate functions are bounded using the known convex and concave envelopes. For complex nonlinear terms appearing in the function $f(\mathbf{x})$, both the Basic Reformulation and αBB Reformulation methods can be applied to the subtree corresponding to the complex term. The resulting constraints will be redundant, but may lead to tighter bounds in cases where the Hessian eigenvalue bounds are poor or the Basic Reformulation produces overly relaxed expressions. This hybrid method retains all disadvantages of the two methods, requiring eigenvalue bounds for the interval Hessian of complex expressions while also increasing the problem size by addition of new constraints and variables.

2.3.1 Hybrid Example

Consider the function $f(x) = x(x^2-1)$. Assume the original bounds for x as $\{x|-1 \leq x \leq 1\}$. The Basic Reformulation results in three new variables and bounds on those variables shown below. The term $w_1 = x^2$ uses x^2 as the convex underestimating function and a secant between $(x^L)^2$ and $(x^U)^2$ as the concave overestimating function. The bilinear term can be

relaxed using the convex and concave envelopes given in Appendix A.

$$\begin{aligned} w_1 &= x^2 \\ w_2 &= w_1 - 1 \\ w_3 &= x \cdot w_2 \end{aligned}$$

$$\begin{aligned} 0 \leq \quad & w_1 \quad \leq 1 \\ -1 \leq \quad & w_2 \quad \leq 0 \\ -1 \leq \quad & w_3 \quad \leq 1 \end{aligned}$$

The underestimating relaxations for $x(x^2-1)$ are plotted in Figure 2 using both the Basic and αBB methods. The two bounds appearing from the Basic Reformulation result from the minimal value of the projection of w_2 onto the x, w_3 plane for the two convex functions used in the relaxation of the bilinear term $x \cdot w_2$. The resulting tightest lower bound on w_3 is a line for $-1 \leq x \leq 0$ and $x^2 - 1$ for $0 \leq x \leq 1$.

For the αBB Reformulation, the Hessian for the function $x(x^2 - 1)$ is $6x$, resulting in $\alpha = 3$ for the original partition. The αBB method clearly produces a loose function relaxation in this case, even when the exact eigenvalue of the Hessian function is known.

Figure 2 also shows the convex lower bounds for the partition $\{x|0 \leq x \leq 1\}$. In this case, $\alpha = 0$, resulting in a tight underestimate for the original function. For this partition, the Basic Reformulation produces a looser function relaxation. In this example, the Basic Reformulation or the αBB Reformulation will produce the tighter relaxation, depending upon the current partition for x. Typically, the best relaxation method will not be known *a priori,* so the combined approach is suggested.

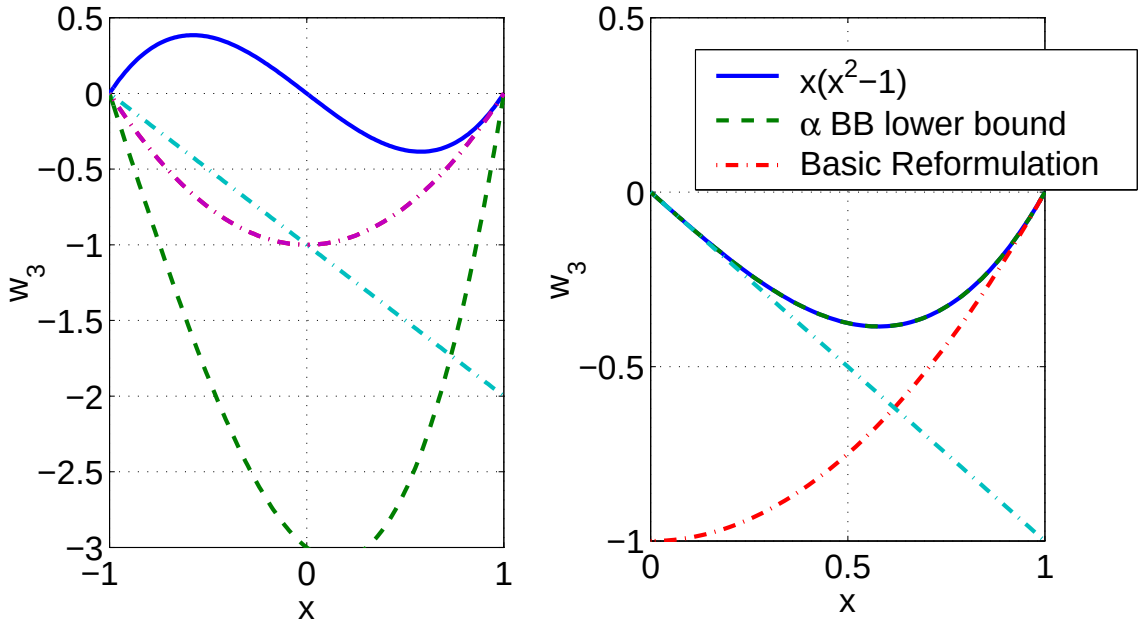

Figure 2: Underestimating relaxations for $x(x^2-1)$ for two different partitions, $-1 \leq x \leq 1$ and $0 \leq x \leq 1$ using the Basic and αBB methods. The αBB lower bound relaxations equals the original function for the second partition.

This example function representation is not unique and could also be coded as $f(x) = x^3 - x$. This alternative representation results in a significantly different relaxation using the Basic Reformulation method. The original representation may be coded by a user in order to reduce the number of terms resulting from multiplication of polynomials. Obviously, the representation of a nonlinear expression chosen by the user may greatly affect the resulting relaxation of the function. This simple scalar function is used to aid the visualization of the problem. The example in Section 2.1.2 demonstrates a complex multivariable expression that is not trivially classified as convex or concave. As a branch-and-bound algorithm progresses, complex nonlinear expressions may be convex or concave for a given partition, resulting in tight bounds using the αBB approach if αBB detects convexity of concavity.

2.4 Advanced Hybrid Reformulation

The Simple Hybrid Reformulation can be extended in order to potentially produce tighter bounds for complex expressions. Using the Basic Reformulation, the αBB method can be applied to complex nonlinear terms appearing in the original nonlinear expressions, as well as application of αBB to intermediate complex nonlinear terms that appear as a result of the Basic Reformulation method. The αBB method should be applied to the nonlinear expression expressed in terms of the original variables $\mathbf{x}$. For intermediate terms with known convex and concave envelopes, the αBB method should not be applied.

Again, the constraints will be redundant but may lead to tighter bounds in some cases, potentially reducing the number of partitions created in a branch-and-bound algorithm. This hybrid method retains all disadvantages of the Basic and αBB methods, but now requires evaluation of eigenvalue bounds for the interval Hessian of many complex expressions.

The example in Section 2.1.2 contains four intermediate terms that may be relaxed using

both methods. The upper bound for the variable w_4 can be calculated exactly using the αBB Reformulation in this example. In other cases, intermediate terms that cannot initially be classified as convex or concave may become purely convex or concave for a smaller partition in a branch-and-bound algorithm, resulting in tight bounds due to the αBB reformulation.

2.5 Linear Reformulation

Given a nonlinear function, the Basic Reformulation method creates new nonlinear and linear constraints describing a convex relaxation of the original function. This set of convex constraints can be further relaxed to a set of linear constraints. This is accomplished by creation of valid outer approximation functions for the convex nonlinear expressions [17].

The convex envelopes resulting from bilinear and linear fractional terms are linear expressions. Other nonlinear terms considered are univariate functions. For monotonic univariate functions, the linear secant function can be used to bound the nonlinear function above or below as appropriate on an interval. In the Basic Reformulation, the nonlinear monotonic function itself is the upper or lower bound for monotonic nonlinear functions. In the Linear Reformulation, valid support functions for nonlinear expressions can be readily derived at the end points on the interval for the function. Additional linearizations of the convex nonlinear functions can be developed at additional points on the interval, tightening the linear relaxation.

The linear bounding functions for the nonlinear functions depend upon the variable bounds. Branch-and-bound procedures proceed by creating partitions of the variable space. When using the Linear Reformulation method for deriving lower bounds on partitions, the new linear relaxation bounding functions must be evaluated for each partition. The nonlinear relationships derived in the Basic Reformulation method can be used to tighten the variable

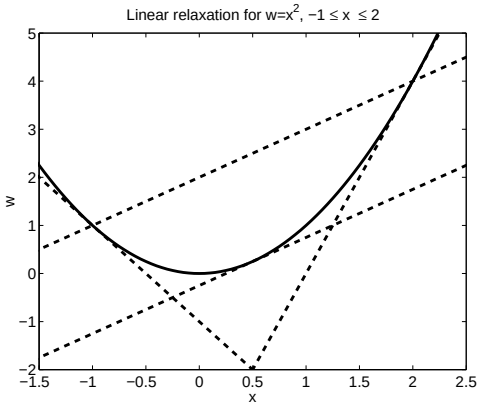

Figure 3: Example linear relaxation for $w = x^2$, $-1 \leq x \leq 2$. The secant serves as the upper bound for the function. Outer approximation supports are developed at end points and an additional single uniformly spaced point.

bounds. Lagrangian based bounds tightening methods [15] are also applicable to the convex linear relaxation problem.

It should be noted that for optimization problems involving both nonconvex and convex nonlinear constraints, both the nonconvex and convex nonlinear constraints are used to derive linear support functions for the linear relaxation reformulation. The convex lower bounding problem becomes a problem involving only linear constraints, some of which depend upon the variable bounds for the current partition. The Linear Reformulation method can also be applied to problems originally containing only convex and linear constraints. The resulting linear relaxation provides a lower bound for the convex nonlinear problem. The solution of the convex problem can be found using branch-and-bound techniques [17].

3 Application

DAEPACK is a software library consisting of symbolic and numeric components for general numerical calculations [19]. A distinguishing feature of DAEPACK is the set of symbolic

components which perform source-to-source transformations on computational codes. For example, given a legacy FORTRAN code for evaluating a model, DAEPACK automatically generates new code for evaluating the analytical partial derivatives of the original model with respect to specified independent variables. This concept, originally developed in the automatic differentiation community, has now been extended to include automatic generation of a larger class of information. The current version of DAEPACK accepts FORTRAN 77 (with some FORTRAN 90 and 95 extensions) and automatically generates portable FORTRAN code. However, DAEPACK has been designed to be readily extended to other procedural programming languages for both the source and target codes.

These new applications include the generation of code for determining the sparsity pattern of a model, construction of a discontinuity-locked model, and the automatic construction of the interval extension of a model. In this paper, these ideas are extended to generate automatically a convex relaxation of a nonlinear model. Specifically, the user provides the original nonlinear nonconvex model coded into a FORTRAN program and DAEPACK automatically generates a collection of new codes which evaluate the convex relaxation of the original model using any of the methods described above: reformulation of the factorable nonconvex functions to simple nonlinear terms, creation of αBB underestimators for complex nonlinear terms, linearization of nonlinear functions, or a hybrid relaxation approach. DAEPACK implements these multiple methods for automated code generation starting from a general constraint formulation, producing portable FORTRAN code which can then be used in any user-developed application.

Optimization applications can significantly benefit from the use of DAEPACK for convexification, interval extension, numerical integration, and automatic differentiation. General nonlinear programming algorithms take advantage of the automated Jacobian evaluation

using automatic differentiation techniques. Nonconvex global optimization algorithms can use a variety of automated convexification methods. αBB based methods can incorporate interval extensions produced by DAEPACK. Dynamic optimization methods can integrate models and evaluate parametric sensitivities using DAEPACK subroutines. DAEPACK delivers the supporting services that enable rapid formulation and solution of nonlinear and mixed-integer optimization problems.

DAEPACK is designed to support the general constructs used in procedural programming languages. Assignment statements, conditional statements, control loops, goto statements, intrinsic functions, and user-supplied subroutine and function calls can be abstracted into this framework. DAEPACK also supports common blocks and general input/output.

Given a FORTRAN source code for a model, the original file is initially translated into internal data structures representing the problem. This first translation step converts the model into an intermediate representation that is essentially independent of the programming language of the original source code. That is, data structures for holding programming constructs such as assignments, loops, procedure calls, and conditional statements are sufficiently general to represent a wide variety of procedural programming languages. In order to extend DAEPACK to handle source code in other programming languages, a translator may be written that converts the desired source into this intermediate representation.

During the next phase, the intermediate representation is converted into a set of data structures used for code transformation and code generation. At this point, common subexpressions appearing in the model are eliminated. This means that if the same expression appears multiple times throughout a model, the subexpression will only introduce one new linear or nonlinear variable into the reformulation. Due to precedence rules, all repeated linear subexpressions are eliminated. Repeated complex nonlinear subexpressions may not

be removed, depending upon the original representation. Future versions of DAEPACK will support limited symbolic reformulation in order to identify additional common subexpressions that would currently remain unexploited.

The user must specify the dependent and independent variables for the problem. With the dependent and independent variable specified, an interprocedural dependency analysis is performed to determine how the independent variables influence the intermediate and dependent variables. Variables that depend on independent variables are referred to as "active" variables, whereas variables that do not are "nonactive" or "passive". This information is used to avoid reformulating sections of code that have no impact on the dependent variables (for example, code which simply computes data in the model).

Symbolic convexification takes place by creating a transformed convex representation of the original problem using one of the procedures described in Section 2. This procedure is straightforward for models involving flat sequences of expressions containing no intermediate variable expressions, loops, conditional statements, or external subroutine calls. In this case, the elementary function representation for each dependent variable expression can be analyzed to identify complex nonlinear and complex intermediate subtree expressions as required by the αBB and hybrid Reformulation methods. The transformation proceeds by recursively reducing the tree expression, producing a new forest of expressions representing the convex relaxation. A source code representation of the convex relaxation new problem can then be generated from this forest.

In more general cases, the internal code generation techniques become more complex. For example, typical codes contain intermediate variables used to avoid repeated computation of the same quantity and simplify large, complex expressions. The dependent variables are then functions of one or more intermediate and/or independent variables. These interme-

diate variables are treated as a part of the expressions describing the dependent variables, which is automatically relaxed using the recursive convexification procedures. New linear or nonlinear variables are introduced during the convex relaxation of the active intermediate variables. These new variables are then used during the convex relaxation of any other expression containing the corresponding intermediate variables. If an intermediate variable is reassigned to a new expression involving active variables then the convex relaxation of the new expression is constructed and the corresponding intermediate variable is assigned to a new linear or nonlinear variable. Care must be taken if intermediate variables are contained within the logical expressions of IF statements present in the code. In this case, the convex relaxation of the intermediate variables may alter the program flow. DAEPACK will replace logical expressions in the IF statements with expressions solely in terms of the original independent variables so that the sequence of statements and their corresponding convex relaxations encountered during execution is the same for both the original code and the convexified code. The presence of conditional statement in the original code is discussed in more detail below.

External user-supplied subroutines and functions that are called within the code are also considered, although they must be available as source in order to be analyzed. There are a number of reasons a programmer will use subroutines and functions within a code, including reducing complexity, increasing reuse, and avoiding duplication of code in several areas of the program. From the standpoint of convex relaxation, these subroutine and function calls simply define additional relationships between the independent and dependent variables. New code will be automatically generated for the user-supplied subroutines and functions for evaluating the convex relaxations of the code defined within them. The interprocedural dependency analysis described above, will identify which of the subroutine or function

arguments are functions of independent variables and, thus, must be convexified. In some cases, the subset of arguments of the subroutine or function that are active will be different at different places the subroutine or function is called. In order to avoid the introduction of unnecessary variables and constraints, several new subroutines or functions will be generated depending on the activity of the subroutine and function arguments. The argument lists of these transformed subroutines or functions are augmented with the additional variables that are introduced within the code during convexification. The outputs of these modified subroutine or function calls are the convex relaxation of the outputs of the original subroutine or function.

Loops within the code may define one or more intermediate and/or dependent variables. These constraints are convexified exactly as they would be if they appeared outside the loop, however, the number of new linear or nonlinear variables introduced are computed as the loop is evaluated. The number of executions of the loop may not be known until run-time. However, by dynamically computing the number of new constraints added during the execution of the loop, the generated code will be correct for any valid input. A special case for the use of a loop is to compute the limit of a convergent sequence. Depending on the initial value for the limit of the sequence, the number of loop iterations performed during a program execution may change from one execution of the loop to the next. Consequently, the number of new variables and constraints introduce will change from call to call. These types of implicit functions are not considered in the convexification methods presented in Section 2. As a result, DAEPACK does not support these types of loops.

Conditional statements can also be considered. Each conditional clause in an IF statement can be considered an atom, taking only Boolean values. Complex conditional expressions can be represented internally using a tree expression involving equality, inequality,

AND, OR, and NOT operators. Using propositional logic techniques [14], new constraints can be written for each subtree expression represented by a Boolean operation. A binary variable can then be used to represent the Boolean value of each subtree in an expression for the current evaluation. Assignment statements within a conditional expression containing dependent or intermediate variable assignments can be treated as normal expressions to be relaxed accordingly. The resulting convex constraints can be enforced or ignored, depending upon the current values of the binary variables.

3.0.1 Logic Example

The user could supply the following code fragment containing a conditional statement:

```
IF ((x(1) .GE. 3).AND.(x(1) .LE. 3)) THEN
x(1)=x(2)**2.0d0
ELSE
x(1)=x(2)**3.0d0
END IF
```

This represents the following logic constraints:

$$\begin{aligned} (x_1 \geq 3) \wedge (x_1 \leq 5) &\Rightarrow x_1 = (x_2)^2 \\ \neg\left((x_1 \geq 3) \wedge (x_1 \leq 5)\right) &\Rightarrow x_1 = (x_2)^3 \end{aligned}$$

The conditional clause contains three atoms, resulting in three new binary variables b_1, b_2, and b_3. First consider expressing the atom $x_1 \geq 3 \Rightarrow b_1$. The following constraints can be

used to represent this relationship and its converse, assuming M takes a large positive value:

$$\begin{aligned} x_1 - 3 &\leq Mb_1 \\ -x_1 + 3 &< M(1 - b_1) \end{aligned}$$

In the previous expression, the second new constraint forcing b_1 to 0 when $x_1 - 3 < 0$ can be written as $-x_1 + 3 + \epsilon \leq M(1 - b_1)$ for $\epsilon \geq 0$. The second expression $x_1 \leq 5 \Rightarrow b_2$ and the overall expression $b_1 \wedge b_2 \Rightarrow b_3$ result in the following constraints:

$$\begin{aligned} 5 - x_1 &\leq Mb_2 \\ -5 + x_1 + \epsilon &\leq M(1 - b_2) \\ b_1 + b_2 - 1 &\leq b_3 \\ 2 - (b_1 + b_2) &\leq 2(1 - b_3) \end{aligned}$$

In cases where $b_3 = 1$, the constraint $x_1 = (x_2)^2$ must be enforced, and when $b_3 = 0$, the constraint $x_1 = (x_2)^3$ must be enforced. This is accomplished with the following four constraints. These constraints can be relaxed using the normal methods.

$$\begin{aligned} x_1 - (x_2)^2 &\leq M(1 - b_3) \\ -x_1 + (x_2)^2 &\leq M(1 - b_3) \\ x_1 - (x_2)^3 &\leq Mb_3 \\ -x_1 + (x_2)^3 &\leq Mb_3 \end{aligned}$$

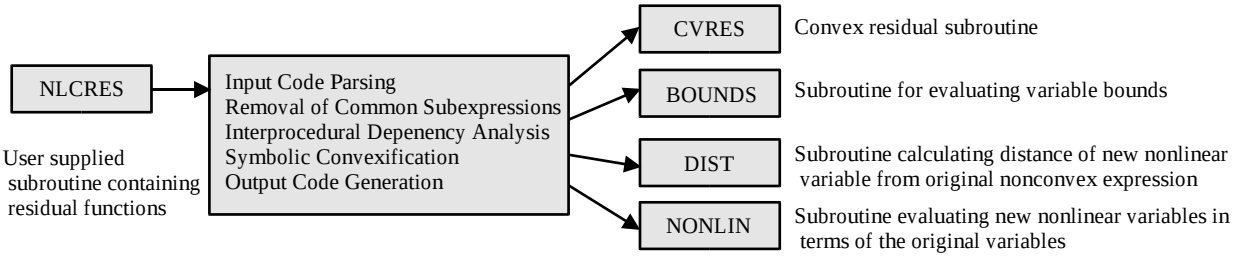

Figure 4: DAEPACK source-to-source conversion of FORTRAN constraint representation

3.1 DAEPACK Example

Given a FORTRAN vector valued residual subroutine, DAEPACK can produce four new source code files for evaluation of a variety of convex model information, as represented in Figure 4. The user provides a file containing a subroutine of residual values for a set of nonlinear expressions, NLCRES. The first automatically generated file, CVRES, contains a new subroutine which returns residual values for the linear version of the original equations, the residual values for the new linear relationships, and the residual functions for the new convex relationships. This routine also provides the number of new linear and nonlinear variables used in the reformulation, as well as the new number of constraints derived from nonconvex expressions. The second file, BOUNDS, contains source code used to evaluate and tighten variable bounds, based upon existing variable bounds. The third file, DIST, contains code that can be evaluated to determine the difference between a new nonlinear variable and the corresponding original nonconvex function. For example, the nonlinear variable w may be introduced for the bilinear term x_1x_2. The distance would be evaluated for the current variable values as $w - x_1x_2$. This information can be used in global optimization routines to determine which nonconvex term is poorly relaxed, providing information useful for deciding on potential branching variables. The final file, NONLIN contains code which returns the value of the new nonlinear variables in terms of original variables. This can readily be

used to determine the dependency of new nonlinear variables on the original variables. This information is also useful for developing Hessian information used in the αBB and Hybrid Reformulation methods.

Figure 5 shows example source code produced by DAEPACK. The resulting source code contains normal FORTRAN code which can be manipulated by DAEPACK for automatic differentiation or interval analysis purposes.

3.2 Branch-and-Bound Application

Applications of DAEPACK in a branch-and-bound algorithm using convexification, automatic differentiation, and interval analysis techniques are shown in Figure 6 for solution of a nonconvex nonlinear programming problem. The constraints of the original problems are currently classified as linear, nonlinear convex, and nonconvex by the user and separated into respective source files LINEAR, CONVEX, and NLRES. This preprocessing is currently accomplished using MAPLE, but will eventually be accomplished with DAEPACK Hessian functionality. DAEPACK automatic differentiation can be used to generate Jacobian information for the nonlinear constraints, generating files CONVEXAD, and NLRESAD. When compiled, subroutines evaluating constraint values in LINEAR, CONVEX, and NLRES along with the analytical Jacobian values obtained from CONVEXAD and NLRESAD can be used to solve a local nonconvex upper bounding problem for a specified branch-and-bound partition.

Convexification of the original nonconvex constraints results in four new subroutines. The Jacobian of the CVRES subroutine yields the linear coefficients for the original nonconvex constraints, the linear coefficients for the new linear relationships, and the Jacobian of the new nonlinear convex constraints. Using this information, a lower bounding con-

Original Code →

```
      subroutine cons(delta,mm,nn,x)
      double precision delta(mm),x(nn)
      integer mm,nn
      delta(1)  =x( 1)−x(13)*((x(25)*x(26)*(x(25)+x(26))*0.5d0)
     c    **−0.3333333333d0)
```

Transformed Code for Convex Residual Evaluation →

```
      subroutine conscv(delta,mm,nn,x,x_lo,x_up,zzz_nlvar,zzz_lvar,
     $   zzz_lvar_lo,zzz_lvar_up,zzz_lcon,zzz_nnvar,zzz_nvar,
     $   zzz_nvar_lo,zzz_nvar_up,zzz_nncon,zzz_ncon,zzz_nrhs)
!!independent { x }
!!dependent   { delta }
      implicit none
      integer nn
      integer mm
      double precision x(nn),x_lo(nn),x_up(nn)
      double precision delta(mm)
      integer zzz_nlvar
      double precision zzz_lvar(zzz_nlvar)
      double precision zzz_lvar_lo(zzz_nlvar),zzz_lvar_up(zzz_nlvar)
      double precision zzz_lcon(zzz_nlvar)
      integer zzz_nnvar
      double precision zzz_nvar(zzz_nnvar)
      double precision zzz_nvar_lo(zzz_nnvar),zzz_nvar_up(zzz_nnvar)
      integer zzz_nncon
      double precision zzz_ncon(zzz_nncon),zzz_nrhs(zzz_nncon)
      double precision zzz_alpha,zzz_tmp
C###  New constraints for convexification of bilinear term #1
      zzz_ncon(1)=x_lo(25)*x(26)+x_lo(26)*x(25)−zzz_nvar(1)
      zzz_nrhs(1)=x_lo(25)*x_lo(26)
      zzz_ncon(2)=x_up(25)*x(26)+x_up(26)*x(25)−zzz_nvar(1)
      zzz_nrhs(2)=x_up(25)*x_up(26)
      zzz_ncon(3)=−x_lo(25)*x(26)−x_up(26)*x(25)+zzz_nvar(1)
      zzz_nrhs(3)=−x_lo(25)*x_up(26)
      zzz_ncon(4)=−x_up(25)*x(26)−x_lo(26)*x(25)+zzz_nvar(1)
      zzz_nrhs(4)=−x_up(25)*x_lo(26)
C###
C###  New linear constraint #1
      zzz_lcon(1)=zzz_lvar(1)−(x(25)+x(26))
C###
C###  New constraints for convexification of bilinear term #2
      zzz_ncon(5)=zzz_nvar_lo(1)*zzz_lvar(1)+zzz_lvar_lo(1)*
     $   zzz_nvar(1)−zzz_nvar(2)
      zzz_nrhs(5)=zzz_nvar_lo(1)*zzz_lvar_lo(1)
      zzz_ncon(6)=zzz_nvar_up(1)*zzz_lvar(1)+zzz_lvar_up(1)*
     $   zzz_nvar(1)−zzz_nvar(2)
      zzz_nrhs(6)=zzz_nvar_up(1)*zzz_lvar_up(1)
      zzz_ncon(7)=−zzz_nvar_lo(1)*zzz_lvar(1)−zzz_lvar_up(1)*
     $   zzz_nvar(1)+zzz_nvar(2)
      zzz_nrhs(7)=−zzz_nvar_lo(1)*zzz_lvar_up(1)
      zzz_ncon(8)=−zzz_nvar_up(1)*zzz_lvar(1)−zzz_lvar_lo(1)*
     $   zzz_nvar(1)+zzz_nvar(2)
      zzz_nrhs(8)=−zzz_nvar_up(1)*zzz_lvar_lo(1)
C###
C###  New linear constraint #2
      zzz_lcon(2)=zzz_lvar(2)−(zzz_nvar(2)*0.5d0)
C###
      if(zzz_lvar_lo(2).ge.0.0d0) then
C###  New constraints for convexification of nonlinear term #3
        zzz_ncon(9)=−zzz_nvar(3)+(zzz_lvar(2))**−0.3333333333d0
        zzz_nrhs(9)=0.0d0
        zzz_ncon(10)=zzz_nvar(3)−(zzz_lvar_up(2)**−0.3333333333d0
     $     −zzz_lvar_lo(2)**−0.3333333333d0)/(zzz_lvar_up(2)−
     $     zzz_lvar_lo(2))*(zzz_lvar(2)−zzz_lvar_lo(2))
        zzz_nrhs(10)=zzz_lvar_lo(2)**−0.3333333333d0
C###
      else
        write(*,*) 'Invalid nonlinear expression.'
        write(*,*) 'Try reformulating problem.'
        stop
      end if
C###  New constraints for convexification of bilinear term #4
      zzz_ncon(11)=x_lo(13)*zzz_nvar(3)+zzz_nvar_lo(3)*x(13)
     $   −zzz_nvar(4)
      zzz_nrhs(11)=x_lo(13)*zzz_nvar_lo(3)
      zzz_ncon(12)=x_up(13)*zzz_nvar(3)+zzz_nvar_up(3)*x(13)
     $   −zzz_nvar(4)
      zzz_nrhs(12)=x_up(13)*zzz_nvar_up(3)
      zzz_ncon(13)=−x_lo(13)*zzz_nvar(3)−zzz_nvar_up(3)*x(13)
     $   +zzz_nvar(4)
      zzz_nrhs(13)=−x_lo(13)*zzz_nvar_up(3)
      zzz_ncon(14)=−x_up(13)*zzz_nvar(3)−zzz_nvar_lo(3)*x(13)
     $   +zzz_nvar(4)
      zzz_nrhs(14)=−x_up(13)*zzz_nvar_lo(3)
C###
      delta(1)=x(1)−zzz_nvar(4)
C###  New constraints for convexification of bilinear term #5
      zzz_ncon(15)=x_lo(26)*x(27)+x_lo(27)*x(26)−zzz_nvar(5)
      zzz_nrhs(15)=x_lo(26)*x_lo(27)
      zzz_ncon(16)=x_up(26)*x(27)+x_up(27)*x(26)−zzz_nvar(5)
      zzz_nrhs(16)=x_up(26)*x_up(27)
      zzz_ncon(17)=−x_lo(26)*x(27)−x_up(27)*x(26)+zzz_nvar(5)
      zzz_nrhs(17)=−x_lo(26)*x_up(27)
      zzz_ncon(18)=−x_up(26)*x(27)−x_lo(27)*x(26)+zzz_nvar(5)
      zzz_nrhs(18)=−x_up(26)*x_lo(27)
C###
C###  New linear constraint #3
      zzz_lcon(3)=zzz_lvar(3)−(x(26)+x(27))
C###
```

Figure 5: Example of original model source code and resulting convex model source code using basic formulation.

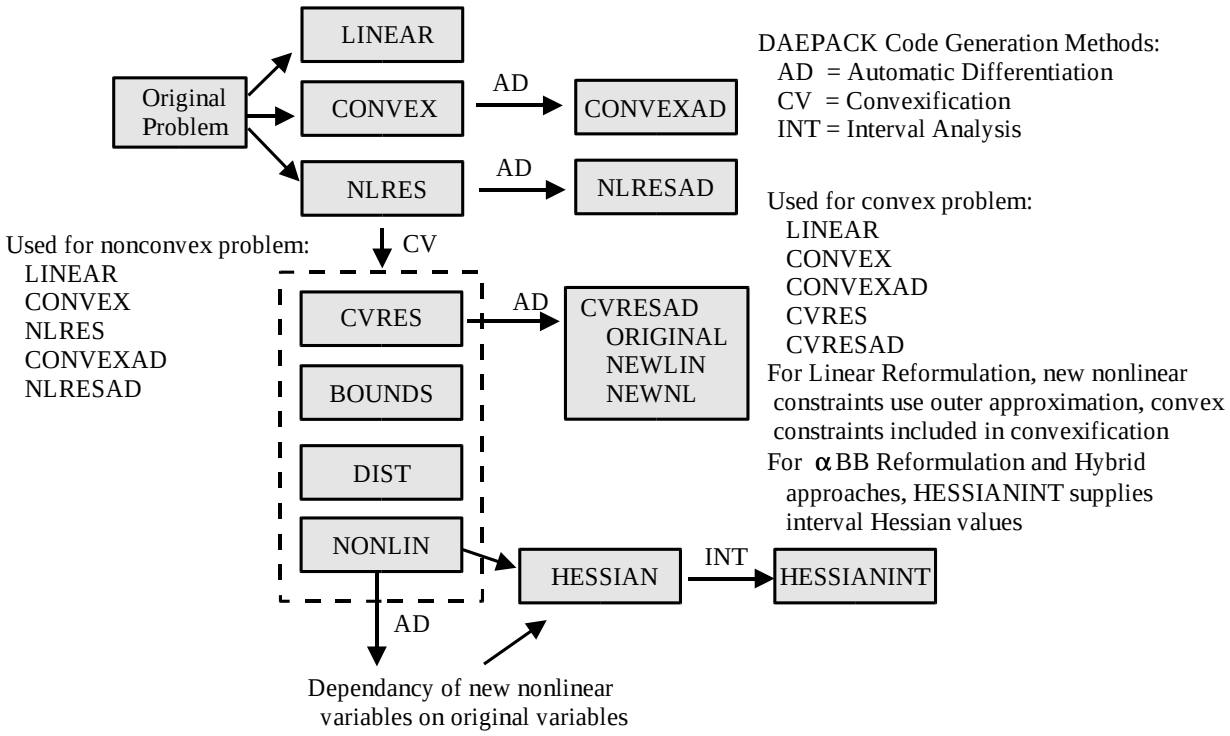

Figure 6: Application of DAEPACK for global solution of nonconvex optimization problems, solving local upper bounding problems and convex lower bounding problems for a given partition.

vex problem can be solved for a given partition in the branch-and-bound algorithm. The BOUNDS subroutine can be used to update variable bounds based on the interval extensions of the nonlinear relationships. The DIST subroutine provides information useful for deciding which nonconvex relationship is poorly approximated in the convex problem. The NONLIN subroutine provides the actual relationships for the new nonlinear variables in terms of the original variables. This can also be used to provide reasonable initial guess for new nonlinear variables, given values of the original variables. The sparsity pattern of this subroutine also provides the relationships for each nonlinear constraint, information useful for deciding on new branching variables.

Using the Linear Reformulation, the new nonlinear convex relations in the file CVRES actually only contain linear constraints derived as outer approximations of the nonlinear convex expressions. The Jacobian values from CVRESAD can be updated based upon the current variable bounds, providing the linear constraint coefficients for a given branch-and-bound partition. For αBB and Hybrid Reformulation, the nonlinear relationships expressed in NONLIN can be used to derive Hessian expressions. The interval extension of source code containing the Hessian elements can be used to bound the eigenvalues of the Hessians.

4 Results

A branch-and-reduce optimization routine has been developed based on [15, 16, 2] to support development of the general purpose outer approximation algorithm described in [10]. This method has been tested on a variety of example from taken from [15, 4]. Bounds tightening techniques based upon interval analysis of nonlinear constraints and linear constraints have been established. Additionally, Lagrangian based techniques can be used for bounds tightening. General optimization problems can be automatically analyzed, generated, and solved

Table 1: Comparison of objective function results using various relaxation techniques for HENS problem on a single partition.

Reformulation Type	Basic	αBB	Simple Hybrid	Advanced Hybrid	Linear 3 Supports	Linear 6 Supports	Linear 10 Supports
Objective 1	112563	104192	112563	120276	111741	112418	112549
Objective 2	134338	125466	134338	137026	133141	134084	134325
Variables	136	72	136	136	136	136	136
Nonlinear Constraints	12	24	36	84	0	0	0
Linear Constraints	249	65	249	249	285	321	369

starting from a simple text file containing the problem formulation.

4.0.1 Lower Bound Comparison

Table 1 shows a comparison of objective function lower bound results for using the various relaxation techniques on the MINLP HENS problem presented in [20, 2, 1]. These problems were derived by fixing the binary variables to the global solution values and solving the resulting lower bounding convex problem on two different variable partitions. The row for Objective 1 corresponds to the relaxation at the root node (first partition), while Objective 2 is the lower bound for a partition potentially encountered in the branch-and-bound search.

NPSOL 5.0 was used to solve the nonlinear convex NLP problems with a tolerance of 1e-6, while CPLEX 7.0 was used for the linear problems. The eigenvalue bounds for αBB and Hybrid methods were calculated using the interval extension of Gerschgorin's theorem [2]. The known concavity of intermediate terms was exploited for the Advanced Hybrid method.

The number of linear outer approximation support functions for this example was examined at values of 3, 6, and 10 evenly spaced intervals. As expected, the Basic method presents a better lower bound than the linear formulations, although the linear formulation with 10 evenly spaced approximations is within the relative tolerance of the nonlinear con-

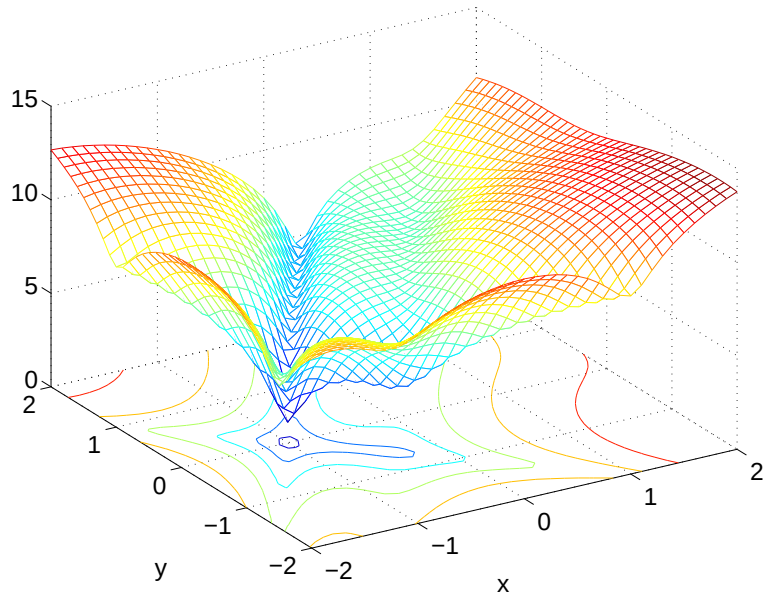

Figure 7: Objective function manifold for example problem for Problem 4.

vex NLP result. Compared to the Basic reformulation, the αBB method lower bound is worse, the Basic Hybrid method shows no improvement, and the Advanced Hybrid method provides a better bound. A variety of methods for bounding the eigenvalues of a symmetric matrix are available, and could easily yield tighter bounds for the αBB and hybrid methods. DAEPACK creates general purpose code for calling eigenvalue bounding subroutines, allowing the user to use the default method or a user developed method.

4.0.2 Global Solution Comparison

The following problem from [7, 4] is considered for global solution. This problem has no constraints, but exhibits significant nonlinearity, as seen in Figure 7.

$$\begin{aligned} \min \quad & \left[1+(x+y+1)^2\left(19-14x+3x^2-14y+6xy+3y^2\right)\right] \\ \times \quad & \left[30+(2x-3y)^2\left(18-32x+12x^2+48y-36xy+27y^2\right)\right] \end{aligned} \tag{4}$$

The global solution occurs at $x = 0$, $y = -1$ with an objective value of 3. The relaxation techniques require that realistic variable limits be specified. Results are presented for $x^L = y^L = -2$ and $x^U = y^U = 2$.

NPSOL 5.0 was used with a relative tolerance of 1e-6 for solution of nonlinear convex lower bounding problems. The absolute tolerance for the global branch-and-bound solution was 1e-2. The upper bound values for each partition were generated by evaluating the objective function at the corners and midpoint of the rectangular partition. At each iteration, Lagrangian values, nonlinear relations, and linear relations were used for bounds tightening. A limit of 15 bounds tightening iterations was enforced per iteration. No optimization based bounds tightening was performed at any point in the algorithm. Branching variable decision was based upon worst constraint, then worst ratio of current bounds to original bounds for constraints with more than one variable. Bisection was used for branchpoint selection. An objective function cut was used in the lower bounding problem, based on the current best upper bound. The αBB constraints were scaled by 1e-4. Gerschgorin's theorem was used for calculation of Hessian eigenvalues. The objective function was expressed as a nonlinear constraint, requiring addition of a single variable and constraint. All runs were performed on a Pentium III 755 MHz processor with 128 MB RAM running Linux 2.2.14.

Table 2 shows a comparison of branch-and-reduce results for Problem 4. The αBB method did not converge after 10,000 iterations. This may be due to calculation of loose lower bounds for Hessian eigenvalues using Gerschgorin's theorem. A variety of other methods are

Table 2: Comparison of branch-and-reduce results using various relaxation techniques for Problem 4.

Reformulation Type	Basic	αBB	Simple Hybrid	Advanced Hybrid	Linear 3 Supports	Linear 10 Supports
Partitions Created	3147	>10000	2625	2128	3265	3255
Convex Problems	7847	>10000	9201	7435	6531	6511
Solution Time (s)	420	*	736	773	150	166
Variables	20	3	136	20	20	20
Nonlinear Constraints	28	2	30	34	0	0
Linear Constraints	9	1	9	9	45	73

*αBB did not converge for this problem after 10,000 iterations.

available for bounding the eigenvalues of a symmetric matrix. Additionally, improved interval bounds on the Hessian elements may be possible using symbolic reformulation techniques. The αBB method converges to the global solution in this implementation when limiting the variable range to the global solution $\pm$1e-4 using 511 partitions in 31 seconds. Table 2 shows the hybrid methods benefit from the use of additional constraints via a reduction the total number of partitions created, although overall solution times are worse than solution using the Basic Reformulation method. Again, improved techniques for bounding Hessian eigenvalues could improve results when using the hybrid methods for convexification.

The linear formulation appears quite promising. As compared to the Basic Reformulation, these results show a slight increase in total number of partitions created while improving the overall solution times. The linear formulation with 10 support functions for each nonlinear expression exhibits little difference in terms of number of partitions as compared to the basic reformulation. The linear method approaches the moderately tight convexification of the Basic method, while exploiting the stability, robustness, scalability, and speed of linear programming techniques.

This simple comparison of convexification techniques is by no means intended to be exhaustive. Obviously, different problems will present different results using various relaxation

methods. Results may also change with application of different heuristics or additional extensions to the current global optimization implementation, including interval tests for constraints and optimization based bounds tightening techniques. This example shows that the hybrid reformulation techniques can potentially reduce the total number of partitions explored in a branch and bound problem. Further, this example demonstrates that DAEPACK can automatically produce the varied information needed by global optimization algorithms using a very general model representation.

5 Conclusions

The symbolic source-to-source functionality of DAEPACK has been extended to create convex problem reformulation given an original source code model. This general purpose tool can be used for development of new global optimization algorithms. DAEPACK produces portable code given a model represented in a general FORTRAN subroutine. A variety of convexification techniques are available, including the Basic Reformulation of McCormick, the αBB Reformulation of Adjiman *et. al*, and the Linear Reformulation of Ryoo and Sahinidis. Hybrid methods have also been proposed for improving the resulting convex relaxation. These relaxation methods have been demonstrated in a branch-and-reduce example, presenting tradeoffs between convexification tightness and computational time for nonlinear relaxation techniques. Relaxation based on linear outer approximation of the Basic Reformulation of McCormick can mitigate this tradeoff, using linear programming technology with moderately tight bounds on the problem relaxation.

Acknowledgement

The authors would like to acknowledge financial support from the National Computational Science Alliance.

References

[1] C. S. Adjiman. *Global Optimization Techniques for Process Systems Engineering.* PhD thesis, Princeton University, Princeton, NJ, 1998.

[2] C. S. Adjiman, S. Dalliwig, C. A. Floudas, and A. Neumaier. A Global Optimization Method, αBB, For General Twice-Differentiable Constrained NLPs - I Theoretical Advances. *Comput. Chem. Eng.*, 22(9):1137–1158, 1998.

[3] M. A. Duran and I. E. Grossman. An Outer-Approximation Algorithm for a Class of Mixed-Integer Nonlinear Programs. *Mathematical Programming*, 36:307–339, 1986.

[4] C. A. Floudas *et. al. Handbook of Test Problems in Local Global Optimization.* Kluwer Academic Publishers, 1999.

[5] J. E. Falk and R. M. Soland. An Algorithm for Separable Nonconvex Programming Problems. *Management Science*, 15(9):550–569, 1969.

[6] R. Fletcher and S. Leyffer. Solving Mixed Integer Nonlinear Programs by Outer Approximation. *Mathematical Programming*, 66:327–349, 1994.

[7] A. Goldstein and J. Price. On Descent from Local Minima. *Mathematics of Computation*, 25:569–574, 1971.

[8] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia, 2000.

[9] D. Hertz, C. S. Adjiman, and C. A Floudas. Two Results on Bounding the Roots of Interval Polynomials. *Comput. Chem. Eng.*, 23:1333–1339, 1999.

[10] P. Kesavan and P. I. Barton. Decomposition Algorithms for Nonconvex Mixed-Integer Nonlinear Programs. *Fifth International Conference on Foundations of Computer-aided Process Design, Breckenridge, CO. AIChE Symposium Series*, 96:323–325, 2000.

[11] G. P. McCormick. Computability of Global Solutions to Factorable Nonconvex Programs: Part I - Convex Underestimating Problems. *Mathematical Programming*, 10:147–175, 1976.

[12] R. E. Moore. *Methods and Applications of Interval Analysis.* SIAM, Philadelphia, 1979.

[13] L. B. Rall. Automatic Differentiation: Techniques and Applications. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science.* 1981.

[14] R. Raman and I. E. Grossman. Integration of Logic and Heuristic Knowledge in MINLP Optimization for Process Synthesis. *Comput. Chem. Eng.*, 16(3):155–171, 1992.

[15] H. S. Ryoo and N. V. Sahinidis. Global Optimization of Nonconvex NLPS and MINLPs With Application to Process Design. *Comput. Chem. Eng.*, 19(5):551–566, 1995.

[16] E. M. B. Smith. *On the Optimial Design of Continuous Processes.* PhD thesis, Imperial College, London, 1996.

[17] M. Tawarmalani and N. V. Sahinidis. Global Optimization of Mixed Integer Nonlinear Programs: A Theoretical and Computational Study. Technical report, 2000.

[18] J. E. Tolsma and P. I. Barton. On Computational Differentiation. *Comput. Chem. Eng.*, 22(4):475–490, 1998.

[19] J. E. Tolsma and P. I. Barton. DAEPACK: An Open Modeling Environment for Legacy Code. *Ind. Eng. Chem. Res.*, 39(6):1826–1839, 2000.

[20] T. F. Yee and I. E. Grossman. Simultaneous Optimization Models for Heat Integration - II. Heat Exchanger Network Synthesis. *Comput. Chem. Eng.*, 14(10):1165–1184, 1990.

6 Appendix A- Convex Envelopes and Bounds of Simple Nonlinear Functions

6.1 Bilinear and Linear Fractional Terms

Bilinear terms appear as $w = x_1 x_2$. Bounds for w and x_1 are expressed in Table 3. Bounds for x_2 are similar to those displayed for x_1. Linear fractional terms appear as $w = \frac{x_1}{x_2}$ and can be reformulated to a bilinear form, $x_2 = w x_2$.

$$\begin{aligned} w &\leq x_1^L x_2 + x_1 x_2^U - x_1^L x_2^U \\ w &\leq x_1^U x_2 + x_1 x_2^L - x_1^U x_2^L \\ w &\geq x_1^L x_2 + x_1 x_2^L - x_1^L x_2^L \\ w &\geq x_1^U x_2 + x_1 x_2^U - x_1^U x_2^U \end{aligned}$$

6.2 Univariate Monotonic Functions

Univariate monotonic functions $w = f(x)$ can be bounded using the secant function. For convex functions, the secant function derived from the bounds on x becomes the upper bound and the function itself serves as a lower bound. For concave functions, the secant function can be used as the lower bound and the function itself serves as a upper bound. Bounds on w can be derived by evaluating the function at x^L and x^U. If the function is invertible, bounds can be derived for x as a function of w^L and w^U.

6.3 Univariate Power terms

6.3.1 Variable Raised to a Constant

There are many possible cases for creating the convex relaxation of equations in the form $w = x^c$, where c is some constant in $\mathbb{R}$ and w, $x \in \mathbb{R}$. A secant function can be used as an upper or lower bound on the function $w = x^c$, depending upon the case.

The special case for $w = x^c$ where c is an odd integer and $0 \in [x^L, x^U]$ should be noted. This function contains both convex and concave portions on the region $[x^L, x^U]$. Secant functions of the following form can be used to bound this function. These bounds converge to $w = x^c$ for $x = 0$. For $0 \notin [x^L, x^U]$, the normal convex envelope functions are used.

$$\begin{aligned} w &\leq 0 + \frac{\left(x^U\right)^c - 0}{x^U - x^L}\left(x - x^L\right) \\ w &\geq \left(x^L\right)^c + \frac{0 - \left(x^L\right)^c}{x^U - x^L}\left(x - x^L\right) \end{aligned}$$

Given upper and lower bounds on w and x, it is also useful to develop bounds on w and x. These bounds are given in the Table 4.

6.3.2 Constant Raised to a Variable

Consider the function $w = c^x$, where c is some strictly positive constant in $\mathbb{R}$ and w, $x \in \mathbb{R}$. A secant function can be used as the upper bound on the function. The function itself serves as the lower bound. The bonds on w and x are given in Table 5.

6.3.3 Variable Raised to a Variable

For the expression $w = (x_1)^{x_2}$, constraint can be reformulated as $\ln(w) = x_2 \ln(x_1)$ and treated as an expression involving monotonic functions and a bilinear expression.

New Bounds on w	New Bounds on x_1 if $\lfloor x_2 \rfloor \geq \epsilon$
$w^L = \max\left\{w^L, \min\left\{x_1^L x_2^L, x_1^L x_2^U, x_1^U x_2^L, x_1^U x_2^U\right\}\right\}$ $w^U = \min\left\{w^U, \max\left\{x_1^L x_2^L, x_1^L x_2^U, x_1^U x_2^L, x_1^U x_2^U\right\}\right\}$	$x_1^L = \max\left\{x_1^L, \min\left\{\frac{w^L}{x_2^L}, \frac{w^L}{x_2^U}, \frac{w^U}{x_2^L}, \frac{w^U}{x_2^U}\right\}\right\}$ $x_1^U = \min\left\{x_1^U, \max\left\{\frac{w^L}{x_2^L}, \frac{w^L}{x_2^U}, \frac{w^U}{x_2^L}, \frac{w^U}{x_2^U}\right\}\right\}$

Table 3: Bounds for w and x_1 given the function $w = x_1 x_2$ with w, x_1, $x_2 \in \mathbb{R}$.

c	x	Secant	New Bounds on w	New Bounds on x
$c > 1,\ c \notin \mathbb{N}$	$0 \leq x^L$	Over	$w^L = \max\left\{w^L, (x^L)^c\right\}$ $w^U = \min\left\{w^U, (x^U)^c\right\}$	$x^L = \max\left\{x^L, (w^L)^{\frac{1}{c}}\right\}$ $x^U = \min\left\{x^U, (w^U)^{\frac{1}{c}}\right\}$
$0 < c < 1$	$0 \leq x^L$	Under	$w^L = \max\left\{w^L, (x^L)^c\right\}$ $w^U = \min\left\{w^U, (x^U)^c\right\}$	$x^L = \max\left\{x^L, (w^L)^{\frac{1}{c}}\right\}$ $x^U = \min\left\{x^U, (w^U)^{\frac{1}{c}}\right\}$
$c = 1$	$x \in [x^L, x^U]$	Linear		
$c = 2n,\ n \in \mathbb{N}$	$0 \in [x^L, x^U]$	Over	$w^L = \max\left\{w^L, 0\right\}$ $w^U = \min\{w^U, \max\{(x^L)^c, (x^U)^c\}\}$	$x^L = \max\left\{x^L, -(w^U)^{\frac{1}{c}}\right\}$ $x^U = \min\left\{x^U, (w^U)^{\frac{1}{c}}\right\}$
	$0 < x^L$	Over	$w^L = \max\left\{w^L, (x^L)^c\right\}$ $w^U = \min\left\{w^U, (x^U)^c\right\}$	$x^L = \max\left\{x^L, (w^L)^{\frac{1}{c}}\right\}$ $x^U = \min\left\{x^U, (w^U)^{\frac{1}{c}}\right\}$
	$x^U < 0$	Over	$w^L = \max\left\{w^L, (x^U)^c\right\}$ $w^U = \min\left\{w^U, (x^L)^c\right\}$	$x^L = \max\left\{x^L, -(w^U)^{\frac{1}{c}}\right\}$ $x^U = \min\left\{x^U, -(w^L)^{\frac{1}{c}}\right\}$
$c = 2n+1,\ n \in \mathbb{N}$	$0 \in [x^L, x^U]$	Both	$w^L = \max\left\{w^L, (x^U)^c\right\}$ $w^U = \min\left\{w^U, (x^L)^c\right\}$	$x^L = \max\left\{x^L, (w^L)^{\frac{1}{c}}\right\}$ $x^U = \min\left\{x^U, (w^U)^{\frac{1}{c}}\right\}$
	$0 < x^L$	Over	$w^L = \max\left\{w^L, (x^U)^c\right\}$ $w^U = \min\left\{w^U, (x^L)^c\right\}$	$x^L = \max\left\{x^L, (w^L)^{\frac{1}{c}}\right\}$ $x^U = \min\left\{x^U, (w^U)^{\frac{1}{c}}\right\}$
	$x^U < 0$	Under	$w^L = \max\left\{w^L, (x^U)^c\right\}$ $w^U = \min\left\{w^U, (x^L)^c\right\}$	$x^L = \max\left\{x^L, (w^L)^{\frac{1}{c}}\right\}$ $x^U = \min\left\{x^U, (w^U)^{\frac{1}{c}}\right\}$
$c = -2n,\ n \in \mathbb{N}$	$0 < x^L$	Over	$w^L = \max\left\{w^L, (x^U)^c\right\}$ $w^U = \min\left\{w^U, (x^L)^c\right\}$	$x^L = \max\left\{x^L, (w^U)^{\frac{1}{c}}\right\}$ $x^U = \min\left\{x^U, (w^L)^{\frac{1}{c}}\right\}$
	$x^U < 0$	Over	$w^L = \max\left\{w^L, (x^L)^c\right\}$ $w^U = \min\left\{w^U, (x^U)^c\right\}$	$x^L = \max\left\{x^L, (w^L)^{\frac{1}{c}}\right\}$ $x^U = \min\left\{x^U, (w^U)^{\frac{1}{c}}\right\}$
$c = -2n+1,\ n \in \mathbb{N}$	$0 < x^L$	Over	$w^L = \max\left\{w^L, (x^U)^c\right\}$ $w^U = \min\left\{w^U, (x^L)^c\right\}$	$x^L = \max\left\{x^L, (w^U)^{\frac{1}{c}}\right\}$ $x^U = \min\left\{x^U, (w^L)^{\frac{1}{c}}\right\}$
	$x^U < 0$	Under	$w^L = \max\left\{w^L, (x^U)^c\right\}$ $w^U = \min\left\{w^U, (x^L)^c\right\}$	$x^L = \max\left\{x^L, (w^U)^{\frac{1}{c}}\right\}$ $x^U = \min\left\{x^U, (w^L)^{\frac{1}{c}}\right\}$

Table 4: Enumeration of possible cases for creation of the convex relaxation of $w = x^c$. Here it is assumed w, $x\,\mathbb{R}$ with c a constant $\in \mathbb{R}$. The secant function can over/under estimate the function, depending upon the case. Additionally, expressions are provided for developing tight bounds on w and x.

New Bounds on w	New Bounds on w
$w^L = \max\left\{w^L, \min\left\{c^{\left(x^L\right)}, c^{\left(x^U\right)}\right\}\right\}$ $w^U = \min\left\{w^U, \max\left\{c^{\left(x^L\right)}, c^{\left(x^U\right)}\right\}\right\}$	$x^L = \max\left\{x^L, \min\left\{\frac{\ln(w^L)}{\ln(c)}, \frac{\ln(w^U)}{\ln(c)}\right\}\right\}$ $x^U = \min\left\{x^U, \max\left\{\frac{\ln(w^L)}{\ln(c)}, \frac{\ln(w^U)}{\ln(c)}\right\}\right\}$

Table 5: Bounds expressions for the function $w = c^x$ where c is some strictly positive constant in $\mathbb{R}$ and w, $x \in \mathbb{R}$.