Detection and Transformation of Second-Order Cone Programming Problems in a General-Purpose Algebraic Modeling Language

Jared Erickson
SAS Institute Inc.
Jared.Erickson@sas.com

Robert Fourer
AMPL Optimization Inc.
4er@ampl.com

May 7, 2019

Abstract. Diverse forms of nonlinear optimization problems can be recast to the special form of second-order cone problems (SOCPs), permitting a wider variety of highly effective solvers to be applied. Popular solvers assume, however, that the necessary transformations to required canonical forms have already been identified and carried out. We describe a general approach to the construction of algorithms that automatically detect equivalent to SOCPs and apply the necessary transformations. To test this approach, the algorithms are implemented in the context of the AMPL modeling language and various solvers. The automated transformations are seen to allow for more effective and reliable modeling, while in some cases transforming problems to forms that are much easier to solve.

This work was supported in part by the National Science Foundation Operations Research program under grant DMI-0800662.

1. Introduction

Algebraic modeling languages for optimization [4, 13, 15] serve as an intermediary between the model formulations that people find natural and intuitive, and the problem instance formats that allow algorithms to operate most efficiently. Their design and implementation is thus most challenging when the distance between the modeler's form and the algorithm's form is greatest. We consider here one particularly challenging class of problems for algebraic modeling languages, that of second-order cone programs (or SOCPs).

A second-order cone can be visualized as the convex region defined by that part of the cone $v_1^2 + \ldots + v_n^2 \leq v_0^2$ where $v_0 \geq 0$. The minimization of a linear function over the intersection of such cones is an SOCP. From a modeler's perspective, the great attraction of this problem class lies in the many functional forms that are reducible to it, and their many applications [16]. SOCPs arise, for example, from reformulations of sums and maxima of norms of affine functions, ratios of quadratic to affine functions, logarithmic Chebychev terms, and products of powers of affine functions, when such expressions appear in appropriate contexts within objectives and constraints.

In contrast, the writer of algorithmic software for solving SOCPs assumes that whatever their origin, they have been put into a canonical form before submission. A good canonical form is not so restricted as to require awkward conversions involving large numbers of auxiliary variables and constraints, but not so general as to greatly slow or complicate the computational process. Thus a solver may require an SOCP to be presented as a linear program augmented by any number of quadratic constraints of the form

$$a_1 v_1^2 + \ldots + a_n v_n^2 \leq a_{n+1} v_{n+1}^2, \quad v_{n+1} \geq 0 \tag{1}$$

and perhaps also the "rotated" form

$$a_1 v_1^2 + \ldots + a_n v_n^2 \leq a_{n+1} v_{n+1} v_{n+2}, \quad v_{n+1} \geq 0, \; v_{n+2} \geq 0 \tag{2}$$

with nonnegative coefficients $a_1, \ldots, a_{n+1}$. Like linear and "elliptic" quadratic constraints ($x^T Q x \leq q$ for positive semidefinite Q), these "conic" constraints can be represented concisely through coefficient lists and handled efficiently by interior-point methods; though in contrast to the elliptic case which is detected numerically, the conic case is identified by the structure of the terms.

For a general-purpose modeling language to effectively support SOCPs, it must be able to express and detect many of the different forms of SOCP-equivalent problems, *and* must be able to transform these to the various canonical forms required by different solvers. In this work we show that a collection of practical algorithms can be built to carry out the requisite detection and transformation. These algorithms operate on arbitrary nonlinear expression graphs, employing a recursive node-processing approach that accommodates not only the basic SOCP-equivalent forms but arbitrarily complex combinations of them. The result is a transformation to a solver-independent canonical form that is readily converted to the forms required by various solvers.

We provide in Section 2 below a summary of the background relevant to this work: the principles underlying representation and processing of nonlinear expressions; the range of expressions that we consider for automatic detection and transformation to SOCPs; and the range of other research and implementation in this

area. Then in the main part of the paper we make a case for carrying out detection and transformation through recursive tree-walks; we have tried to make the presentation detailed enough to explain the challenges that we addressed in this work, but not so detailed as to involve the reader in the minutiae of implementation. Sections 3 and 4 introduce the main issues using the relatively simple cases of sum-of-norms objectives and general conic quadratic constraints, after which Section 5 shows what is involved in a more complex case based on a generalization of the geometric mean. Section 6 briefly mentions the full generality of the cases we considered, and section 7 describes the routine we use in many of the cases to check nonnegativity of affine terms.

Finally in section 8 we describe results of applying an implementation, based on the AMPL modeling language and several solvers, to some well-known sets of nonlinear optimization test problems. We observe that SOCP-transformable problems are reasonably common and that the transformations can make problems easier to solve.

2. Background

To set the context and terminology for our work, we begin by reviewing the general approach that has been adopted by modeling languages to detect and transform structures in algebraic expressions. Then we present the range of structures that we propose to detect and to transform to SOCPs. Prototypes of these structures are well known [16], but our taxonomy is more comprehensive, reflecting our goal of detecting SOCP-equivalent expressions in whatever forms might be most convenient for a modeler to write. We conclude this section by briefly surveying the treatment of SOCP expressions in existing modeling software.

2.1 Representation and processing of expression structures

Whereas canonical SOCPs can be represented by lists of linear and quadratic coefficients, most of the nonlinear optimization problems equivalent to SOCPs must be communicated in a more general way. In this work, we focus on objectives and constraints that can be written using mathematical operators and functions, and so can be conveniently represented in the form of *expression graphs.* As an example (which can appear in certain SOCP-transformable objectives and constraints), the mathematical expression

$$\sqrt{(x_1+2)^2+(x_2+1)^2}+(2x_1^2+5x_2^2)/y$$

has the expression graph depicted in Figure 1. Each operator and function in the expression is represented by a node in the graph, with its children corresponding to its operands or arguments. Variables and constants are the terminal nodes. Any node together with its descendents thus describes a subexpression, in the same way that the whole graph beginning with the root (top) node describes the entire expression. The expression graph can always be regarded without loss of generality as a tree such as shown in our diagram, though for efficiency it may be stored as a directed acyclic graph in which all nodes for the same variable, constant, or even subexpression are merged.

Algebraic modeling languages use expression trees to communicate nonlinear objective and constraint expressions to solvers. Continuing our example, a general

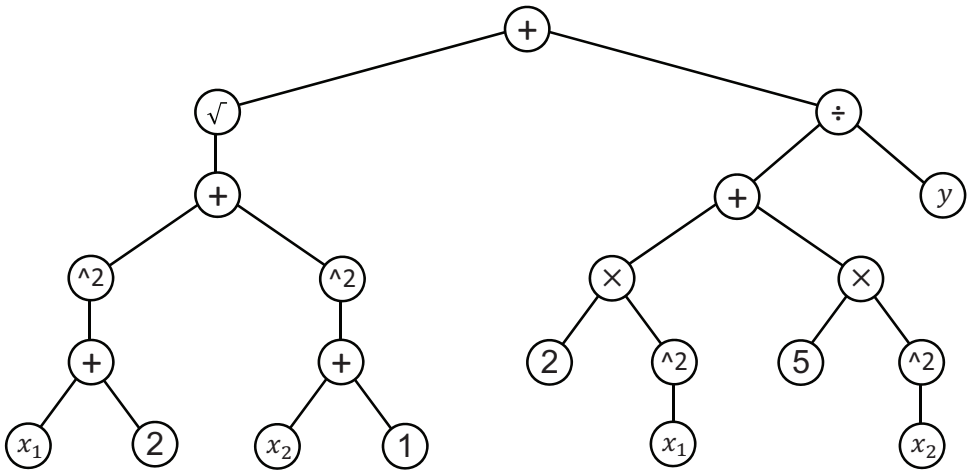

Figure 1. *Expression graph for* $\sqrt{(x_1+2)^2+(x_2+1)^2}+(2x_1^2+5x_2^2)/y$.

algebraic representation of the same expression in the AMPL language [7] could be a symbolic objective function defined as

```
minimize SumNormRatio:
   sqrt(sum {i in S} (x[i]+a[i])^2) + (sum {j in T} b[j]*x[j]^2)/y;
```

The AMPL processor instantiates this expression with particular values for sets (`S`, `T`) and parameters (`a`, `b`) and translates it to a tree data structure suitable for communication and processing [10]; Figure 1 is the tree that results from one possible instantiation. As a result of the widespread use of this approach, there is already considerable experience with analyzing and transforming tree representations to meet solver requirements. Of greatest importance to our work, algorithms for many kinds of operations can be naturally described and efficiently implemented as procedures that "walk" the tree, performing an appropriate action at each node after recursively walking the trees rooted at its children [9].

To test for a certain property in an algebraic expression, a recursive tree-walking function can be implemented directly from a listing of the algebraic operations that characterize the property. As a simple example, operations that produce quadratic expressions include sums and multiples of quadratics, as well as products and squares of linears; Figure 2 gives a systematic list of these operations (in the form to be used by this paper). The corresponding function `isQuadratic` takes a node as its argument and returns `true` if an only if that node describes a subexpression that is quadratic; Figure 3 shows how it might be described in simple pseudocode.

The function `isQuadratic`, together with analogous, simpler functions `isLinear` and `isConstant`, provides an implementation of a detection routine for quadratic expressions. Given a program variable, say `Root`, that points to the root node of an expression tree of interest, an implementation can test quadraticity by calling `isQuadratic(Root)`; appropriate recursive calls are triggered to analyze descendant nodes, with the return value being `True` or `False` according to whether the represented expression can be recognized as quadratic. This simple idea can be applied to detect any structure characterized by a list of rules; for example its application to convexity is described in [8].

QUADRATIC
Sum: $e_1 + e_2$ is QUADRATIC *if* e_1, e_2 are QUADRATIC
Product: $e_1 e_2$ is QUADRATIC *if*
e_1 is QUADRATIC and e_2 is CONSTANT or
e_1 is CONSTANT and e_2 is QUADRATIC or
e_1, e_2 are LINEAR
Square: l^2 is QUADRATIC *if* l is LINEAR
Quotient: e_1/e_2 is QUADRATIC *if* e_1 is QUADRATIC and e_2 is CONSTANT
Variable: v is QUADRATIC *if* v is VARIABLE
Constant: c is QUADRATIC *if* c is CONSTANT

Figure 2. *Detection of a quadratic expression: List of operations.*

```
Boolean isQuadratic (node)
  case node.type:
    Sum:
      return isQuadratic (node.lft) and isQuadratic (node.rgt);
    Product:
      return (isQuadratic (node.lft) and isConstant (node.rgt)) or
             (isConstant (node.lft) and isQuadratic (node.rgt)) or
             (isLinear (node.lft) and isLinear (node.rgt));
    Square:
      return isLinear (node.lft);
    Quotient:
      return isQuadratic (node.lft) and isConstant (node.rgt);
    Variable: Constant:
      return True;
  end case;
end isQuadratic;
```

Figure 3. *Detection of a quadratic expression: Recursive tree-walk function.*

This same idea can be extended moreover to create recursive tree-walking routines for transformation of an expression tree. This is the approach used to convert trees for quadratic expressions into lists of quadratic terms and their coefficients, as required by specialized solvers.

The detection and transformation procedures described in this paper apply a conceptually similar approach, but in much more challenging contexts. Any instantiation of the AMPL objective function `SumNormRatio` shown above must be detected as being capable of transformation to an appropriate SOCP form, for example, and then a transformation involving definition of auxiliary variables and construction of additional linear and quadratic constraints must be carried out.

In general, to process an optimization problem, we must first apply our full range of detection routines to each nonlinear objective and constraint expression tree. If detection is successful we then apply to each expression tree an appropriate transformation routine, corresponding to the structure that was detected. At the end, all nonlinear expressions are quadratic and can be converted to coefficient lists as the solvers require.

2.2 Structures to be detected and transformed

The class of optimization problems transformable to SOCPs has been subject to extensive study, as surveyed for example by Lobo *et al.* [16] and by Ben-Tal and Nemirovski [2]. Our interest is focused more narrowly on SOCP-transformable problems that are amenable to expression-tree manipulation, namely those that can be written algebraically using common operators. Within this class, however, we address many variations that, while mathematically equivalent, may pose different challenges for detection and transformation.

In the following review, we start with several quadratic cases that generalize the forms of (1) and (2). Then we address the so-called SOC-representable functions; these include a variety of non-quadratic structures in objectives and constraints, as well as their combinations through sum, max, and positive multiples. Finally we consider non-SOC-representable objective forms that can be handled through the solution of related SOCPs.

Quadratic problems. Certain basic quadratic constraint forms can be recognized by a recursive walk of the expression tree as being transformable to SOCPs. Constraints of the form

$$\sum_{i=1}^{n} a_i(\mathbf{f}_i\mathbf{x} + g_i)^2 \leq a_{n+1}(\mathbf{f}_{n+1}\mathbf{x} + g_{n+1})^2 \tag{3}$$

are precisely (1) together with the linear constraints $v_i = \mathbf{f}_i\mathbf{x}+g_i$ for $i = 1, \ldots, n+1$, provided that $a_1, \ldots, a_{n+1} \geq 0$ and that $\mathbf{f}_{n+1}\mathbf{x} + g_{n+1} \geq 0$ holds for all feasible $\mathbf{x}$. The latter condition is nontrivial to confirm but can be checked efficiently in some practical ways as we discuss in Section 7. Similarly, constraints of the form

$$\sum_{i=1}^{n} a_i(\mathbf{f}_i\mathbf{x} + g_i)^2 \leq a_{n+1}(\mathbf{f}_{n+1}\mathbf{x} + g_{n+1})(\mathbf{f}_{n+2}\mathbf{x} + g_{n+2}) \tag{4}$$

are precisely (2) together with the linear constraints $v_i = \mathbf{f}_i\mathbf{x}+g_i$ for $i = 1, \ldots, n+2$, provided that $a_1, \ldots, a_{n+1} \geq 0$ and that $\mathbf{f}_{n+1}\mathbf{x} + g_{n+1} \geq 0$, $\mathbf{f}_{n+2}\mathbf{x} + g_{n+2} \geq 0$ hold for all feasible $\mathbf{x}$.

It is possible to consider a more general approach to quadratic constraints, by observing that any collection of inequalities among quadratic and linear expressions can be put into the form $x^TQx+q^Tx+d \leq 0$, using the detection and transformation approach described earlier in this section. If Q has only one negative eigenvalue — as it obviously does for the special case of (1) — then transformation to the form of (1) may be possible; necessary and sufficient conditions for the existence of such a transformation are described by Mahajan and Munson [18]. The detection and transformation of constraints that satisfy these conditions involves, however, a numerical factorization of Q, and thus is distinct from the cases we consider in this paper, which rely on symbolic analyses of the expression tree.

SOC-representable problems. A function $SOC(\mathbf{x})$ is SOC-representable [16] if $SOC(\mathbf{x}) \leq \mathbf{f}_{n+1}\mathbf{x} + g_{n+1}$ can be represented equivalently by some collection of second-order cone constraints (1) and (2) plus possibly some linear constraints. Any positive multiple, sum, or maximum of SOC-representable functions is also SOC-representable; this recursive definition is readily incorporated within the recursive

tree-walk algorithms that we employ. Minimizing any SOC-representable function $f(\mathbf{x})$ is also equivalent to an SOCP, since it can be transformed to minimizing v_{n+1} subject to $f(\mathbf{x}) \leq v_{n+1}$.

Considering the quadratic cases, it is clear that the SOC-representable functions include 2-norms $\|\mathbf{F}\mathbf{x} + \mathbf{g}\|$ and more generally

$$\sqrt{\sum_{i=1}^{n} a_i(\mathbf{f}_i\mathbf{x} + g_i)^2}$$

from taking the square root of both sides of (3), and quadratic-linear ratios of the form

$$\frac{\sum_{i=1}^{n} a_i(\mathbf{f}_i\mathbf{x} + g_i)^2}{\mathbf{f}_{n+2}\mathbf{x} + g_{n+2}} \tag{5}$$

from dividing the nonnegative denominator into both sides of (4).

The negative geometric mean, $-\prod_{i=1}^{p}(\mathbf{f}_i\mathbf{x} + g_i)^{1/p}$, $\mathbf{f}_i\mathbf{x} + g_i \geq 0$, is known to be SOC-representable [1]. Indeed we consider detecting a generalization of this function,

$$-\prod_{i=1}^{p}(\mathbf{f}_i\mathbf{x} + g_i)^{\alpha_i}$$

for exponents satisfying $\sum_{i=1}^{p} \alpha_i \leq 1$, as well as the reciprocal form

$$\prod_{i=1}^{p}(\mathbf{f}_i\mathbf{x} + g_i)^{-\alpha_i}$$

Both are SOC-representable for rational $\alpha_i \geq 0$ and $\mathbf{f}_i\mathbf{x} + g_i \geq 0$.

The p-norm, $(\sum_{i=1}^{n} |\mathbf{f}_i\mathbf{x} + g_i|^p)^{1/p}$, $p \geq 1$, is also known to be SOC-representable [1]. Here again we consider a more general SOC-representable form,

$$(\sum_{i=1}^{n} a_i|\mathbf{f}_i\mathbf{x} + g_i|^{\alpha_i})^{1/\alpha_0}$$

where we require only rational exponents $\alpha_i \geq \alpha_0 \geq 1$, and $a_1, \ldots, a_n \geq 0$. Under the same conditions there are related SOCP equivalents for constraints of the form

$$\sum_{i=1}^{n} a_i|\mathbf{f}_i\mathbf{x} + g_i|^{\alpha_i} \leq a_{n+1}(\mathbf{f}_{n+1}\mathbf{x} + g_{n+1})^{\alpha_0}$$

where $\mathbf{f}_{n+1}\mathbf{x} + g_{n+1} \geq 0$, as well as objectives of the form

$$\text{Minimize} \quad \sum_{i=1}^{n} a_i|\mathbf{f}_i\mathbf{x} + g_i|^{\alpha_i}$$

For $\alpha_0 = 1$ and all $\alpha_i = 2$ these latter two cases can be multiplied out to yield elliptic quadratic constraints and objectives. Their alternative transformations to SOCPs may be useful however when they are combined by sum or max with other SOC-representable forms.

Other minimization problems. SOCP conversions are also known in the case of a more general form of the product-of-powers function,

$$\text{Maximize} \quad \prod_{i=1}^{n}(\mathbf{f}_i\mathbf{x} + g_i)^{\alpha_i}$$

for rational $\alpha_i > 0$ and $\mathbf{f}_i\mathbf{x} + g_i \geq 0$, and for the "logarithmic Chebychev" function

$$\text{Minimize} \quad \max_{i=1}^{n} |\log(\mathbf{f}_i\mathbf{x}) - \log(g_i)|$$

These do not have corresponding constraint forms, as the transformation to an SOCP changes the objective's function value (though not the optimal set).

2.3 SOCPs in other modeling systems

Several modeling languages and systems have been designed or extended to permit representation of second-order cone programs and equivalent problems, with recognition and transformation being supported to varying degrees. All of these systems interface to a variety of SOCP solvers.

GAMS [3] relies on the modeler to identify SOCP constraints, to convert them to an elementary canonical form, and to identify them in the model. For example in the GAMS constraint

```
w =C= sum (i, v(i));
```

the special relational operator `=C=` is employed to state that the square of variable `w` is greater than or equal to the sum of the squares of the variables `v(i)` — a constraint of the form (1), though with unit coefficients. The rotated form (2) is handled similarly. This design takes advantage of existing linear syntax to streamline the implementation, at the cost of inventing a nonstandard operator and placing responsibility for detection and transformation on the creator of the model.

YALMIP [17], built on the MATLAB scientific programming language, can convert sum, max, and multiple of norms expressions to conic formulations. YALMIP also defines `cone` and `rcone` operators to model conic constraints using an approach similar to that of GAMS. The expressions recognized and converted by YALMIP are limited to ones that "look" convex, however, so that (3), (4), and many of the more general forms described in the previous section are not handled directly.

CVX [11, 12], also built on MATLAB, detects and solves convex optimization problems, including many that it transforms to SOCP forms for solution. It recognizes a broad variety of convex functions, though with a certain amount of detection and reformulation required on the user's part. For example, CVX requires (5) to be reformulated using a special function `quad_over_lin` that specifies numerator and denominator as separate arguments and implicitly constrains the denominator to be positive. In contrast, our goal is to apply SOCP detection and transformation directly to the functional form of (5) and to then confirm independently that nonnegativity of the denominator is implied by the problem constraints.

3. Sum-of-Norms Objective Detection and Transformation

To establish terminology and demonstrate the feasibility of recursive tree walks for SOCP detection and transformation, we begin with the simple example of minimizing a weighted sum of norms,

$$\underset{\mathbf{x}}{\text{Minimize}} \quad \sum_{i=1}^{m} a_i \|\mathbf{F}_i\mathbf{x} + \mathbf{g}_i\|$$

with $\mathbf{F}_i$ being an $n_i \times p$ matrix, and $\mathbf{g}_i$ and $\mathbf{x}$ being vectors of length n_i and p. As there is not a specific norm operator in AMPL, we consider detecting objective functions that have the following more explicit form:

$$\underset{\mathbf{x}}{\text{Minimize}} \quad \sum_{i=1}^{m} a_i \sqrt{\sum_{j=1}^{n_i} (\mathbf{f}_{ij}\mathbf{x} + g_{ij})^2} \tag{6}$$

We call this kind of expression a SumOfNorms, and the expression inside the square root a SumOfSquares, and define these forms recursively as shown in Figure 4.

SumOfNorms
Sum: $e_1 + e_2$ is SumOfNorms *if* e_1, e_2 are SumOfNorms
Product: ce or ec is SumOfNorms *if*
e is SumOfNorms and c is PosConstant
Square root: $\sqrt{e}$ is SumOfNorms *if* e is SumOfSquares

SumOfSquares
Sum: $e_1 + e_2$ is SumOfSquares *if* e_1, e_2 are SumOfSquares
Product: ce or ec is SumOfSquares *if*
e is SumOfSquares and c is PosConstant
Square: e^2 is SumOfSquares *if* e is Linear
Constant: c is SumOfSquares *if* c is PosConstant

Figure 4. *Detection of a sum-of-norms objective.*

These definitions are in fact more general than the mathematical formulas that motivated them. They incorporate multiplication by constants in a recursive way that cannot readily be expressed in a single formula; and they accept a constant as a special case of a squared linear term. In a mathematical context such trivialities are ignored, but in the context of structure recognition it is desirable to recognize as broad a variety of trivially equivalent forms as possible. The user can then write expressions however they are most convenient, rather than having to rearrange them to comply with arbitrary rules, such as on the placement of constants.

Indeed our definition can be extended to other potentially useful possibilities, such as division of a SumOfNorms or SumOfSquares by a positive constant. Extension to a maximum of a sum of norms is also straightforward. And corresponding definitions are straightforwardly introduced for maximization of the negative of a sum of norms, after which subtraction operators and negative constants can be introduced. The cases seen in Figure 4 are nevertheless sufficient to illustrate the essentials of the approach that we take.

Corresponding to the definitions of SumOfNorms and SumOfSquares we can apply the ideas of Section 2 to construct boolean-valued functions `isSumOfNorms` and `isSumOfSquares`. Applied to the root of an expression tree for an objective, `isSumOfNorms` returns `true` if and only if the tree represents a sum of norms in the sense of the SumOfNorms definition. Once a sum-of-norms objective has been detected in this way, the challenge remains to describe a procedure for making a second recursive tree walk that constructs the equivalent SOCP.

Conceptually the equivalent SOCP for a sum of norms (6) can be written as

$$\begin{array}{lll} \text{Minimize} & \sum_{i=1}^{m} a_i y_i & \\ \text{Subject to} & \sum_{j=1}^{n_i} z_{ij}^2 \le y_i^2,\ y_i \ge 0 & i = 1, \ldots, m \\ & z_{ij} = \mathbf{f}_{ij}\mathbf{x} + g_{ij}, & i = 1, \ldots, m;\ j = 1, \ldots, n_i \end{array} \tag{7}$$

A procedure for making this transformation naturally consists of two recursive functions analogous to the detection ones — TransformSumOfNorms and Trans-

formSumOfSquares — which will encounter all of the same cases but which will react to them by building up the transformed problem. The operations of building the transformed problem will include adding linear terms to the objective, defining second-order cone constraints and adding squared terms to the left-hand sides of those constraints, and defining linear constraints.

Schematic descriptions of the transformation functions for this purpose are shown in Figure 5. A function is represented as a list of cases corresponding to different possible forms of the first argument, a subexpression `e`. Within each case is pseudocode for the actions to be taken when that case is encountered; these can include recursive calls, calls to other functions, and actions that create or augment various model components.

Our terminology is chosen to emphasize each function's relationships to the mathematical formulation (7), even though they deal in fundamentally different entities. For example, whereas y_i in the formulation are decision variables, `yi` in TransformSumOfNorms is a program variable denoting an object that represents a decision variable. Where the program is shown as "squaring" this variable and "adding" it to a constraint, it is actually performing the operation of appending a quadratic term to the constraint by adding the appropriate coefficient to the

TransformSumOfNorms (`Expr e, Obj o, real k`)
Sum: `e1 + e2` where `e1`, `e2` are SumOfNorms
    TransformSumOfNorms(`e1,o,k`)
    TransformSumOfNorms(`e2,o,k`)
Product: `c * e1` or `e1 * c` where `e1` is SumOfNorms and `c` is PosConstant
    TransformSumOfNorms(`e1,o,c * k`)
Square root: `sqrt(e1)` where `e1` is SumOfSquares
    `yi :=` NewNonnegVar(); `o += k * yi`
    `qi :=` NewLeCon(); `qi += -yi^2`
    TransformSumOfSquares(`e1,qi,1`)

TransformSumOfSquares (`Expr e, LeCon qi, real k`)
Sum: `e1 + e2` where `e1`, `e2` are SumOfSquares
    TransformSumOfSquares(`e1,qi,k`)
    TransformSumOfSquares(`e2,qi,k`)
Product: `c * e1` or `e1 * c` where `e1` is SumOfSquares and `c` is PosConstant
    TransformSumOfSquares(`e1,qi,c * k`)
Square: `sqr(zij)` where `zij` is Variable
    `qi += k * zij^2`
Square: `sqr(e1)` where `e1` is Linear
    `zij :=` NewVar(); `qi += k * zij^2`
    `lij :=` NewEqCon(); `lij += zij - e1`
Constant: `c` is PosConstant
    `zij :=` NewVar(); `qi += k * zij^2`
    `lij :=` NewEqCon(); `lij += zij - sqrt(c)`

Figure 5. *Transformation of a sum-of-norms objective.*

constraint data structure. And whereas y_i is a separate decision variable for each index i, there is only one program variable `yi` that — for the purposes of this simple example — can be reused in creating each quadratic constraint.

Given an expression tree that has been identified as representing a sum-of-norms objective, our procedure is applied by passing to TransformSumOfNorms a pointer to the tree's root, along with a new, initially empty objective:

```
o := NewObj();
TransformSumOfNorms(root,o,1)
```

TransformSumOfNorms then works its way recursively down the tree, accumulating in its third argument the effects of multiplication by constants. Each time that it reaches a norm term — signalled by a square root — it appropriately extends the objective and adds a constraint. Specifically, the actions

```
yi := NewNonnegVar(); o += k * yi
```

create another nonnegative variable `yi` and add to the existing objective that variable multiplied by the appropriate constant. Then the actions

```
qi := NewLeCon(); qi += -yi^2
```

create another less-than-or-equal-to constraint and subtract the new variable's square from the (initially empty) left-hand side, producing the beginnings of a quadratic constraint $\ldots \leq y_i^2$ in the SOCP (7). Finally TransformSumOfSquares is called to process the sum of squares inside the square root.

TransformSumOfSquares similarly works its way down the tree, until it reaches a squared linear term. In the general case, it adds a new squared variable to the constraint previously set up for it (by TransformSumOfNorms) and creates a new constraint equating the variable to the linear expression. Special cases are recognized for efficiency (when the linear term is just a variable) and for convenience (when a positive constant is treated as the square of a trivial linear term).

This approach, constructing recursive definitions and procedures for detection and transformation, can be extended to all of the SOCP-transformable forms defined in Section 2.2. To illustrate the complexities involved, we next describe two more challenging cases: a generalization of quadratic cone constraints in Section 4, and generalized geometric means in Section 5. Other possibilities are treated briefly in Section 6, and are described in detail in [5]. The heuristic that we have used in most of the detection routines to test nonnegativity of affine terms is presented in Section 7. Finally, Section 8 describes the results of applying implementations of our SOCP detection routines to a large number of test problems, and makes some observations on the effect of the SOCP transformations on efficiency and reliability of solvers.

4. Quadratic Cone Constraint Detection and Transformation

Constraints pose greater challenges than objectives for SOCP detection and transformation, as they involve distinct forms of expressions on the left and on the right of the $\leq$ or $\geq$ operator. Here we illustrate constraint processing with the detection of a generalized, but still mostly quadratic, second-order cone constraint.

In this case the transformation to canonical form is relatively straightforward, so we can focus on the difficulties of detecting a form that is sufficiently general for practical application.

Our generalization begins with the observation that both (3) and (4) are the same on the left-hand side,

$$\sum_{i=1}^{n} a_i(\mathbf{f}_i\mathbf{x} + g_i)^2 \leq \ldots$$

while on the right there is either a squared linear term,

$$\ldots \leq a_{n+1}(\mathbf{f}_{n+1}\mathbf{x} + g_{n+1})^2$$

or a product of linear terms,

$$\ldots \leq a_{n+1}(\mathbf{f}_{n+1}\mathbf{x} + g_{n+1})(\mathbf{f}_{n+2}\mathbf{x} + g_{n+2})$$

with the usual requirement for nonnegativity of $a_1, \ldots, a_{n+1}$ and $\mathbf{f}_{n+1}\mathbf{x} + g_{n+1}$, $\mathbf{f}_{n+2}\mathbf{x} + g_{n+2}$. We can thus look first at generalizations to the left-hand and right-hand sides separately, and can then consider generalizations affecting the overall constraint function.

On the left, equivalence to a canonical SOCP is preserved when the sum of squares is generalized to a maximum of such sums:

$$\max_{i=1,\ldots,m} a_i \sum_{j=1}^{n_i} a_{ij}(\mathbf{f}_{ij}\mathbf{x} + g_{ij})^2 \leq \ldots$$

Indeed this equivalence generalizes to arbitrarily nested combinations of sums-of-squares by addition, maximum, and positive scalar multiplication on the left-hand side. Such a form is not easily expressed in mathematical notation but is accommodated quite naturally by a recursive detection routine, which will serve as a building block for our routine to detect quadratic cone constraints. We can further allow for the use of subtraction, minimum, and negative scalar multiplication by introducing a routine for detection of the negative of the same form. These routines are outlined as PosMaxSumOfSquares and NegMinSumOfSquares in Figure 6; their definitions are entirely analogous except for the case of a linear term squared, which appears only in the "positive" routine.

On the right, the $a_{n+1}(\mathbf{f}_{n+1}\mathbf{x} + g_{n+1})^2$ form remains a quadratic cone when $\mathbf{f}_{n+1}\mathbf{x} + g_{n+1} \leq 0$ rather than ≥ 0. The $a_{n+1}(\mathbf{f}_{n+1}\mathbf{x} + g_{n+1})(\mathbf{f}_{n+2}\mathbf{x} + g_{n+2})$ form remains a quadratic cone in several additional cases: when both terms are ≤ 0; when the constant is negative and the terms are of opposite sign; or when one of the terms is either strictly positive or strictly negative (as then the nonnegativity of the left-hand side forces the other term to have the proper sign).

A further generalization allows the constraint to be written as a $\geq$ with the two sides reversed. When the AMPL processor converts symbolic constraint statements to explicit constraint instances, however, it does not maintain a distinction between the left-hand and right-hand sides. Rather the terms involving variables are moved to one side, any constant terms are moved to the other, and each constraint is sent to the solver as a single expression along with either a finite upper-bound constant (denoting a $\leq$ constraint) or a finite lower-bound constant (denoting a $\geq$ constraint).

PosMaxSumOfSquares
Maximum: $\max(e_1, \ldots, e_n)$ *if* every e_i is PosMaxSumOfSquares
Sum: $e_1 + \cdots + e_n$ *if* every e_i is PosMaxSumOfSquares
Difference: $e_1 - e_2$ *if*
e_1 is PosMaxSumOfSquares and e_2 is NegMinSumOfSquares
Negative: $-e$ *if* e is NegMinSumOfSquares
Product/Quotient: $c_1 e_1$ or $e_1 c_1$ or e_1/c_1 *if*
e_1 is PosMaxSumOfSquares and c_1 is PosConstant or
e_1 is NegMinSumOfSquares and c_1 is NegConstant
Square: l^2 *if* l is Linear
Constant: c *if* c is PosConstant

NegMinSumOfSquares
Minimum: $\min(e_1, \ldots, e_n)$ *if* every e_i is NegMinSumOfSquares
Sum: $e_1 + \cdots + e_n$ *if* every e_i is NegMinSumOfSquares
Difference: $e_1 - e_2$ *if*
e_1 is NegMinSumOfSquares and e_2 is PosMaxSumOfSquares
Negative: $-e$ *if* e is PosMaxSumOfSquares
Product/Quotient: $c_1 e_1$ or $e_1 c_1$ or e_1/c_1 *if*
e_1 is NegMinSumOfSquares and c_1 is PosConstant or
e_1 is PosMaxSumOfSquares and c_1 is NegConstant
Constant: c *if* c is NegConstant

Figure 6. *Detection of a generalized sum of squared linear terms.*

As a result, by the time that an SOCP-equivalent problem reaches our detection routines, it may consist of positive squared terms and one negative quadratic term, or negative squared terms and one positive quadratic term. Moreover either of these forms appearing as a subexpression can be converted by negation, subtraction, or multiplication by a negative constant to the opposite form. Thus we define two detection routines as shown in Figure 7. We test a constraint by applying PosQuadrCone when there is a finite upper-bound constant ≤ 0, and applying NegQuadrCone when there is a finite lower-bound constant ≥ 0.

In a constraint expression that is a NegQuadrCone, there is a lone positive term, and so there are cases for both squared linear terms and products of linear terms. But in a constraint expression that is a PosQuadrCone, there is a lone negative term, and hence only cases for products of linear terms appear; when a negative squared linear term appears in a PosQuadrCone, it is recognized as a subtraction, negation, or negative constant multiple of the corresponding NegQuadrCone case.

Once a PosQuadrCone or NegQuadrCone has been identified, the transformation to canonical form is straightforward. Indeed the same recursive transformations are applied to a PosQuadrCone and to any PosMaxSumOfSquares within it: variables are substituted for linear terms, constant multipliers are distributed over sums, and each maximization is replaced by a new variable together with SOC constraints making that variable $\geq$ each maximand. The complete example of TransformPosQuadrCone is shown in Figure 8. Given an expression tree

PosQuadrCone
Sum: $e_1 + \cdots + e_n$ *if* for some k,
e_k is PosQuadrCone and e_i, $i \neq k$ are PosMaxSumOfSquares
Difference: $e_1 - e_2$ *if*
e_1 is PosQuadrCone and e_2 is NegMinSumOfSquares or
e_1 is PosMaxSumOfSquares and e_2 is NegQuadrCone
Negative: $-e_1$ *if* e_1 is NegQuadrCone
Product: $c_1 e_1$ or $e_1 c_1$ *if*
e_1 is PosQuadrCone and c_1 is PosConstant or
e_1 is NegQuadrCone and c_1 is NegConstant
Product: $l_1 l_2$ or $l_2 l_1$ *if*
l_1 is NegLinear or l_2 is PosLinear or
l_1 is NonPosLinear and l_2 is NonNegLinear
Constant: c *if* c is PosConstant

NegQuadrCone
Sum: $e_1 + \cdots + e_n$ *if* for some k,
e_k is NegQuadrCone and e_i, $i \neq k$ are NegMinSumOfSquares
Difference: $e_1 - e_2$ *if*
e_1 is NegQuadrCone and e_2 is PosMaxSumOfSquares or
e_1 is NegMinSumOfSquares and e_2 is PosQuadrCone
Negative: $-e_1$ *if* e_1 is PosQuadrCone
Product: $c_1 e_1$ or $e_1 c_1$ *if*
e_1 is NegQuadrCone and c_1 is PosConstant or
e_1 is PosQuadrCone and c_1 is NegConstant
Product: $l_1 l_2$ or $l_2 l_1$ *if*
l_1 is PosLinear or l_2 is NegLinear or
l_1 is NonNegLinear and l_2 is NonNegLinear or
l_1 is NonPosLinear and l_2 is NonPosLinear
Square: l^2 *if* l is NonNegLinear or l is NonPosLinear
Constant: c *if* c is NegConstant

Figure 7. *Detection of a generalized quadratic cone constraint.*

```
TransformPosQuadrCone (Expr e, LeCon q, real k)
Sum: sum {i in I} ei
    for {i in I}
      TransformPosQuadrCone(ei,q,k)
Difference: e1 - e2
    TransformPosQuadrCone(e1,q,k)
    TransformNegQuadrCone(e2,q,k)
Negative: -e1
    TransformNegQuadrCone(e,q,k)
Product: e1 * c1 or c1 * e1
    if e1 is PosQuadrCone
      then TransformPosQuadrCone(e1,q,c1 * k)
      else TransformNegQuadrCone(e1,q,c1 * k)
Product: l1 * l2 or l2 * l1
    z1 := NewVar(); z2 := NewVar(); q += -k * z1 * z2
    r1 := NewEqCon(); r1 += z1 + l1
    r2 := NewEqCon(); r2 += z2 - l2
Square: sqr(l)
    z := NewVar(); q += k * z^2
    r := NewEqCon(); r += z - l
Maximum: max {i in I} ei
    z := NewVar(0); q += k * z^2
    for {i in I}
      qi := NewLeqCon(); qi += -z^2
      TransformPosQuadrCone(ei,qi,1)
Constant: c
    z := NewVar(0); q += k * z^2
    r := NewEqCon(); r += z - sqrt(c)
```

Figure 8. *Transformation of a generalized quadratic cone constraint.*

that has been identified by PosQuadrCone as representing a generalized quadratic cone constraint function, we pass to TransformPosQuadrCone a pointer to the tree's root, and a new, initially empty $\leq$ constraint with the same upper bound:

```
q := NewLeCon(ub);
TransformPosQuadrCone(root,q,1)
```

Details of TransformNegQuadrCone in the negative case are entirely analogous.

5. Generalized Geometric Mean Detection and Transformation

To illustrate processing a form that is very different from quadratic, we turn next to the detection and transformation of products of rational powers of nonnegative affine terms. Specifically we consider SOC-representable constraints of the form

$$-\prod_{i=1}^{p}(\mathbf{f}_i\mathbf{x}+g_i)^{\alpha_i} \leq \mathbf{f}_{p+1}\mathbf{x}+g_{p+1}$$

for rational exponents satisfying $\sum_{i=1}^{p} \alpha_i \leq 1$ and $\mathbf{f}_i\mathbf{x} + g_i \geq 0$ for $i = 1, \ldots, p$. Here the need to keep track of the sum of exponents poses additional challenges, but our general approach continues to work well.

Since AMPL records the linear part of this constraint in coefficient lists separate from the expression tree, detection can focus on the product-of-powers term. The recursive tree-walk involves similar algorithms for four interrelated forms: a product having positive powers, its reciprocal product having negative powers, and the negatives of these two. Tree-walk functions for the positive-power variants are shown in Figure 9. In the case that these functions return TRUE they also return a value α equal to the sum of the exponents in the product detected. This complication is of course necessitated by the requirement that the positive exponents sum to at most 1. There is no way to tell from looking any part of an expression whether this condition will be satisfied; and because we admit expressions such as $(x_1^2 x_2^2)^{1/5}$ there is also no way to tell from looking any part of an expression whether this condition will *not* be satisfied. Rather the sum of exponents must be propagated to the top of the expression tree where $\alpha \leq 1$ can be checked.

A practical test for rationality of the exponents must also be supplied. Although technically any number represented in the computer is rational, the number of variables generated in the transformation can be kept manageable only if the denominators of the exponents are of reasonable size. Thus to implement RATIONAL in POSPRODPOSPOW we use a variant of the well-known continued fraction procedure [14] shown in Figure 10. The tolerances `eps1` and `eps2` can be set to regulate

POSPRODPOSPOW(α)
Negative: $-e$ *if* e is NEGPRODPOSPOW(α)
Product: $e_1 e_2$ *with* $\alpha = \alpha_1 + \alpha_2$ *if*
  e_1 is POSPRODPOSPOW(α_1) and e_2 is POSPRODPOSPOW(α_2) or
  e_1 is NEGPRODPOSPOW(α_1) and e_2 is NEGPRODPOSPOW(α_2)
Quotient: e_1/e_2 *with* $\alpha = \alpha_1 - \alpha_2$ *if*
  e_1 is POSPRODPOSPOW(α_1) and e_2 is POSPRODNEGPOW(α_2) or
  e_1 is NEGPRODPOSPOW(α_1) and e_2 is NEGPRODNEGPOW(α_2)
Power: e^{α_2} *with* $\alpha = \alpha_1 \alpha_2$ *if*
  e is POSPRODPOSPOW(α_1) and $\alpha_2 \geq 0$ is RATIONAL or
  e is POSPRODNEGPOW(α_1) and $\alpha_2 \leq 0$ is RATIONAL
Linear: l *with* $\alpha = 1$ *if* l is POSLINEAR

NEGPRODPOSPOW(α)
Negative: $-e$ *if* e is POSPRODPOSPOW(α)
Product: $e_1 e_2$ or $e_2 e_1$ *with* $\alpha = \alpha_1 + \alpha_2$ *if*
  e_1 is POSPRODPOSPOW(α_1) and e_2 is NEGPRODPOSPOW(α_2)
Quotient: e_1/e_2 *with* $\alpha = \alpha_1 - \alpha_2$ *if*
  e_1 is POSPRODPOSPOW(α_1) and e_2 is NEGPRODNEGPOW(α_2) or
  e_1 is NEGPRODPOSPOW(α_1) and e_2 is POSPRODNEGPOW(α_2)
Linear: l *with* $\alpha = 1$ *if* l is NEGLINEAR

Figure 9. *Detection of a generalized SOC-representable geometric mean.*

```
RATIONAL (real frac, num, den)
num := 0; den := 1; den_prv := 1;
repeat
  if frac - floor(frac) < eps1 return True
  if abs(frac - num / den) < eps2 return False
  frac := 1 / (frac - floor(frac))
  den_nxt := den * floor(frac) + den_prv
    den_prv := den; den := den_nxt
  num := round(frac * den)
```

Figure 10. *Rationality test for a fraction between 0 and 1.*

what is accepted as rational; our tests used 10^{-6} for both.

After successful detection, for transformation purposes the constraint may be regarded as

$$y \leq \prod_{i=1}^{p} z_i^{\alpha_i}$$

where all $\alpha_i \geq 0$ and $\sum_{i=1}^{p} \alpha_i \leq 1$. Auxiliary linear constraints define $y = -(\mathbf{f}_{p+1}\mathbf{x} + g_{p+1})$ and $z_i = \mathbf{f}_i\mathbf{x} + g_i$ for any affine term that is not simply a variable.

Since the exponents are rational they can be written as $\alpha_i = \beta_i/M$ for a suitably chosen common integer denominator and integer numerators satisfying $L = \sum_{i=1}^{p} \beta_i \leq M$. Thus the constraint can equivalently be written

$$y^M \leq \prod_{i=1}^{p+1} z_i^{\beta_i}$$

with dummy variable $z_{p+1} = 1$, where we make $\sum_{i=1}^{p+1} \beta_i = M$ by taking $\beta_{p+1} = M - L$. Finally we change the power M on the left to the next-highest power of 2, by multiplying both sides of the constraint by y^{N-M} where $N = \lceil 2^{\log_2 M} \rceil$. Then the constraint has the equivalent form

$$y^N \leq \prod_{i=1}^{p+2} z_i^{\beta_i}$$

where $z_{p+2} = y$ and $\beta_{p+2} = N - M$, giving $\sum_{i=1}^{p+2} \beta_i = N$.

At this point if $N = 2$ then this is already a standard or rotated cone constraint, while otherwise we can break it into a rotated cone constraint plus two similar constraints of the same form but with half the exponent:

$$y^2 \leq w_1 w_2, \;\; w_1^{N/2} \leq \prod_{i=1}^{p_1} z_i^{\beta_{1i}}, \;\; w_2^{N/2} \leq \prod_{i=1}^{p_2} z_i^{\beta_{2i}}$$

with $\sum_{i=1}^{p_1} \beta_{1i} = \sum_{i=1}^{p_2} \beta_{2i} = N/2$. The same transformation is applied recursively until the exponents are reduced to 2 and all the constraints are conic.

```
ExtractProduct (Expr e, IndexSet G, real k, alpha)
Negative: -e1
    TransformProduct(e1,G,k,alpha)
    k := -k
Product: c1 * e1 or e1 * c1
    TransformProduct(e1,G,k,alpha)
    k := k * c1^alpha
Product: e1 * e2
    TransformProduct(e1,G,k,alpha)
    TransformProduct(e2,G,k,alpha)
Quotient: e1 / e2
    TransformProduct(e1,G,k,alpha)
    TransformProductInv(e2,G,k,alpha)
Power: e1 ^ c1
    TransformProduct(e1,G,k,c1*alpha)
Linear: l1
    z := NewVar(); G := G union {z}; Rational(alpha,z.num,z.den)
    r := NewEqCon(); r += z - l1
```

Figure 11. *Transformation of an SOC-representable product of positive powers.*

As this description suggests, our transformation algorithm must first perform a recursive tree-walk to collect the variables and exponents in $\prod_{i=1}^{p} z_i^{\alpha_i}$. This is done by the routine ExtractProduct in Figure 11. This routine can in fact be used by all of the product-of-powers transformations, as the appropriate detection routine has already determined that the result will be valid. The input parameter `alpha` carries exponent values down the tree to the individual product terms; the output parameter `k` returns constant multipliers that are aggregated into one constant multiplier of the entire expression. To handle quotients there is an analogous routine TransformProductInv which is not shown.

The results of the tree-walk are used in TransformPosProdPow, shown in Figure 12, to recursively break the detected product into SOC constraints. The rational exponent values from TransformProduct are used to compute the constants `L`, `M`, `N`, after which the appropriate `N` variables are set up in advance. The function SplitProduct then recursively splits the original constraint into simple rotated-cone constraints, using the new variables at the lowest level.

6. Further Generalizations for SOC-Representable Forms

In addition to the examples of the preceding sections, we have implemented detection and transformation routines for the other cases described in Section 2.2. All are included in the tests to be described in Section 8. We comment briefly in this section on two further generalizations that can be built upon the SOC-representable cases.

First, any sum, maximum, or positive multiple of SOC-representable forms is also recognized as SOC-representable. Thus it is possible to transform various objectives

```
TransformPosProdPow (Expr e)
G := Empty; k := 1
TransformProduct(e,G,k,1)

M := lcm {z in G} z.den
for {z in G}
  z.beta := (z.num/z.den) * M
L := sum {z in G} z.beta
N := 2 ^ ceil(log2(M))

y := NewVar(); u := NewVar(1,1); i := 0
for {z in G}
  Z[i+1] := ... := Z[i+z.beta] := z
  i := i + z.beta
Z[L+1] := ... := Z[M] := u
Z[M+1] := ... := Z[N] := y

SplitProduct (y,0,0,k)

SplitProduct (Var y, Int lev, Int pos, Real const)
q := NewLeCon();
if lev = log2(M) - 1
  q += y^2 -k * Z[pos+1] * Z[pos+2]
  pos += 2
else
  w1 := NewVar(); w2 := NewVar();
  q += y^2 - k * w1 * w2
  SplitProduct(w1,lev+1,pos,1)
  SplitProduct(w2,lev+1,pos,1)
```

Figure 12. *Transformation of an SOC-representable product of positive powers.*

and constraints that combine expressions from different SOC-representable cases. This extension has been included in our tested implementation.

Second, it is possible to extend our analysis to generalized rotated cone constraints of the form

$$\sum_{i=1}^{n} a_i f_i(\mathbf{x})^2 \leq a_{n+1} f_{n+1}(\mathbf{x}) f_{n+2}(\mathbf{x})$$

where $a_i \geq 0$ for $i = 1, \ldots, n+1$; $|f_i(\mathbf{x})|$ is SOC-representable for $i = 1, \ldots, n$; and $-f_{n+1}(\mathbf{x})$, $-f_{n+2}(\mathbf{x})$ are SOC-representable with $f_{n+1}(\mathbf{x}) \geq 0$, $f_{n+2}(\mathbf{x}) \geq 0$. This observation permits a straightforward further extension of our detection and transformation algorithms; however it was not implemented for the reported tests.

7. Checking Nonnegativity of Affine Terms

Most of the SOCP forms that we detect involve affine terms that must be nonnegative. In principle, the question of whether $\mathbf{f}_i\mathbf{x} + g_i \geq 0$ for all feasible $\mathbf{x}$ can be

resolved by minimizing $\mathbf{f}_i\mathbf{x} + g_i$ over the problem constraints, but to carry out this minimization in general is much too expensive.

As a practical alternative, nonnegativity can be tested over only the bounds $l_j \le x_j \le u_j$ on the variables, which are accumulated in AMPL and passed to the solver interface separately from the more general constraints. In the simplest case, if $\mathbf{x} \ge 0$ then $\mathbf{f}_i \ge 0$ and $g_i \ge 0$ suffice to imply $\mathbf{f}_i\mathbf{x} + g_i \ge 0$. More generally, given bounds $l_j \le x_j \le u_j$, nonnegativity of $\mathbf{f}_i\mathbf{x} + g_i$ is implied by nonnegativity of the lower bound

$$\sum_{f_{ij}>0} f_{ij}l_j + \sum_{f_{ij}<0} f_{ij}u_j + g_i$$

which is inexpensive to check, particularly in the common case where many of the coefficients f_{ij} are zero and can be omitted from the calculations.

In our implementation, we use a stronger version of this test, which incorporates the bounds and one linear constraint, but is still reasonably fast to compute. For each constraint $\mathbf{f}'_i\mathbf{x} + g'_i \ge 0$, we seek to prove $\mathbf{f}_i\mathbf{x} + g_i \ge 0$ by finding an $\alpha \ge 0$ for which

$$\mathbf{f}_i\mathbf{x} + g_i \ge \alpha(\mathbf{f}'_i\mathbf{x} + g'_i)$$

For $\alpha = 0$ this is the previous bounds test, and for $\alpha = 1$ and $\mathbf{x} \ge 0$ this subsumes the simple test $\mathbf{f}_i \ge \mathbf{f}'_i$, $g_i \ge g'_i$. But more generally, applying the lower-bound formula previously given, this inequality will hold if

$$\sum_{f_{ij}>\alpha f'_{ij}} (f_{ij} - \alpha f'_{ij})l_j + \sum_{f_{ij}<\alpha f'_{ij}} (f_{ij} - \alpha f'_{ij})u_j + (g_i - g'_i) \ge 0$$

The expression to be tested here is a concave piecewise-linear function of α, with breakpoints at $f_{ij}/f'_{ij} > 0$. Thus to test nonnegativity for all α, it is only necessary to evaluate the expression at the breakpoint values. Some special handling is also necessary at $\alpha = 0$, and possibly at the highest breakpoint value if the bound is negative at that point but is continuing to increase.

In practice most of the f_{ij} and f'_{ij} are zero, and some (especially upper) bounds may be infinite, so the number of breakpoints at which the bound must actually be evaluated tends to be small. An efficient implementation is described in [5] and was used for the tests that we report in the next section.

8. Tests of Detection and Transformation

We carried out two kinds of tests to assess the potential usefulness of the ideas proposed in this paper.

First, as we describe in Section 8.1, we applied the detection routines to a large number of examples found in public sets of nonlinear AMPL test problems. The results suggest that SOCP-transformable problems are reasonably common and encompass a variety of cases.

Second, as we describe in Section 8.2, we constructed some test problems to highlight the advantages in solution efficiency that might be realized through the kinds of transformations that we have implemented. Our tests confirm the existence

of problems that are hard to solve in their original formulations, but that become much easier after transformation to equivalent SOCPs.

8.1 Detection tests

As a detection test set we used the 1238 continuous nonlinear optimization problems from the following AMPL model collections:

- `www.orfe.princeton.edu/~rvdb/ampl/nlmodels/`
- `www.netlib.org/ampl/models/nlmodels/`

We tried each problem as input directly to a convex quadratic solver* and as input to our detection tests. Those successfully recognized can be partitioned as follows:

- 66 are numerically recognized by the solver as elliptic quadratic problems.
- 21 are symbolically recognized by the solver and by our tests as conic quadratic problems. These are problems that have the forms (1) and (2) or simple variations thereof.
- 80 are symbolically recognized by our tests as transformable to conic quadratic problems, but could not be recognized by the solver.

The remaining problems are not recognized by the solver or by our tests; the great majority fail to be SOCP-transformable due to the presence of sine functions, equality constraints, and other obviously disqualifying features, though a small number might be convertible to SOCPs by transformations we did not consider.

Overall about 13.5% of the test problems are solvable as convex quadratics, in one way or another. Moreover the problems detected as SOCP-solvable after transformation exhibit a variety of forms, as suggested by the following examples:

- `hs064` has $4/x_1 + 32/x_2 + 120/x_3 \leq 1$
- `hs036` minimizes $-x_1x_2x_3$
- `hs073` has $1.645\sqrt{0.28x_1^2 + 0.19x_2^2 + 20.5x_3^2 + 0.62x_4^2} \leq \ldots$
- `hs049` minimizes $(x_1 - x_2)^2 + (x_3 - 1)^2 + (x_4 - 1)^4 + (x_5 - 1)^6$
- `emfl_nonconvex` has $\sum_{k=1}^{2}(x_{jk} - a_{ik})^2 \leq s_{ij}^2$

These results suggests that varied SOCP-solvable forms occur naturally in applications and can be caught by detection routines more sophisticated than what the solvers employ.

8.2 Transformation tests

To give an idea of the usefulness of SOCP transformations, we compared solving nonlinear optimization problems in their original formulations to solving equivalent canonical SOCP problems produced by our transformation routines. For the original nonlinear problems we used Knitro's three algorithms for nonlinearly constrained nonlinear optimization, and for the equivalent SOCPs we used CPLEX.†

*We observed essentially identical results using the CPLEX 12.5.1 and Gurobi 5.5. Their detection routines for convex quadratic problems have not changed significantly in later versions.

†Our tests used Knitro version 8.1.1. More recently Knitro has added its own specialized algorithm for canonical SOCP problems.

```
var x {1..5} integer;
var y {1..5} >= 0;

minimize obj: sum {i in 1..5} (
   sqrt( (x[i]+2)^2 + (y[i]+1)^2 ) +
   sqrt( (x[i]+y[i])^2 ) + y[3]^2 );

subj to xsum: sum {i in 1..5} x[i] <= -12;
subj to ysum: sum {i in 1..5} y[i] >= 10;
```

Figure 13. *Sum-of-norms test problem.*

For the collections of nonlinear test problems used in Section 8.1, the computation times were too small or the structures too simple to permit much difference to be observed. In these cases, the advantage of the transformation would lie in the convenience of being able to apply popular linear-quadratic solvers to these problems, rather than in any substantial computational efficiency.

Established collections of nonlinear test problems may however tend towards smooth formulations most appropriate to gradient-based nonlinear solvers such as Knitro. Many of the formulations transformable to SOCPs involve functions that have singular or nondifferentiable points. To show that transformation of such functions can make a substantial difference to performance, we constructed the sum-of-norms problem shown in Figure 13, which involves square root functions not differentiable at zero.

None of the three Knitro algorithms found a recognizably optimal point for this problem; the two interior-point algorithms reported "Current feasible solution estimate cannot be improved," and the active-set algorithm reported "Relative change in feasible solution estimate $<$ xtol." After transformation to a conic quadratic, Knitro's interior-point algorithms were able to report a "Locally optimal solution" with the faster of the two requiring 35 iterations and 37 function evaluations. After some further transformations to put the conic problem into canonical form, CPLEX was able to report a globally optimal solution using 10 barrier iterations that involve no function evaluation mechanism.

Similar results were observed with variable `x` required to be integer. The best results in Knitro were with the active-set algorithms, but while Knitro found appropriate integer values, it required much more effort to get convergence of the continuous `y` values; what took 2 seconds in Knitro took only 0.06 seconds in CPLEX.

As a further test of the potential performance advantages of the SOCP transformations in the integer case, we constructed instances having nonlinearities of the same kinds as in four of the examples cited in Section 8.1, but with the variables being integer and much larger in number. Table 1 provides a summary of statistics for these expanded instances, and computational results are summarized in Table 2. The original formulations are seen to be ill-suited to general-purpose nonlinear methods, whereas after transformation the problems are solved quite efficiently as canonical SOCPs.

Instance	Variables	Constraints Linear	Nonlinear	Objective Type
`hs064.mod`	3/21	0/0	1/10	Nonlinear
`hs036.mod`	3/30	1/15	0/0	Nonlinear
`hs073.mod`	4/100	1/50	1/25	Linear
`hs049.mod`	5/500	2/2	0/0	Nonlinear

Table 1. *Statistics for expanded instances. Variable and constraint counts are shown as original / expanded.*

Instance	Transformation + CPLEX	Original formulation in Knitro
`hs064.mod`	28 seconds	Reaches node limit with an optimality gap of 14% in 728 seconds
`hs036.mod`	107 seconds	Finds a "locally optimal solution" in 4 seconds that is 10% worse than optimal
`hs073.mod`	< 1 second	Reaches node limit, with many gradient evaluation errors along the way, in 3173 seconds with an optimality gap of 24.7%
`hs049.mod`	3 seconds	Cannot find a provably optimal solution in 40 minutes

Table 2. *Results for expanded instances. All CPLEX runs reported globally optimal solutions in the times shown.*

References

[1] F. Alizadeh and D. Goldfarb, "Second-order cone programming." *Mathematical Programming* **95** (2003) 3–51.

[2] A. Ben-Tal and A. Nemirovski, *Lectures on Modern Convex Optimization: Analysis, Algorithms, and Engineering Applications*. Society for Industrial and Applied Mathematics (2001).

[3] A. Brooke, D. Kendrick and A. Meeraus, *GAMS: A User's Guide, Release 2.25*. Scientific Press (1992). `www.gams.com/dd/docs/bigdocs/GAMSUsersGuide.pdf`.

[4] B. Dominguez-Ballesteros, G. Mitra, C. Lucas and N.-S. Koutsoukis, "Modelling and Solving Environments for Mathematical Programming (MP): A Status Review and New Directions." *Journal of the Operational Research Society* **53** (2002) 1072–1092.

[5] Jared Erickson, "Detection and Transformation of Objective and Constraint Structures for Optimization Algorithms." Ph.D. Dissertation, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL, USA (2013).

[6] R. Fourer, D.M. Gay and B.W. Kernighan, "A Modeling Language for Mathematical Programming." *Management Science* **36** (1990) 519–554.

[7] R. Fourer, D.M. Gay and B.W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming,* 2nd edition. Duxbury Press, Pacific Grove, CA (2002). See also `www.ampl.com/BOOK/download.html`.

[8] R. Fourer, C. Maheshwari, A. Neumaier, D. Orban and H. Schichl, "Convexity and Concavity Detection in Computational Graphs: Tree Walks for Convexity Assessment." *INFORMS Journal on Computing* **22** (2010) 26–43.

[9] R. Fourer and D. Orban, "DrAmpl: A Meta Solver for Optimization Problem Analysis." *Computational Management Science* **7** (2010) 437-463.

[10] D.M. Gay, Hooking Your Solver to AMPL. Technical Report 97-4-06, Computing Sciences Research Center, Bell Laboratories (1997); at `www.ampl.com/REFS/hooking2.pdf`.

[11] M. Grant and S. Boyd. "Graph Implementations for Nonsmooth Convex Programs." In *Recent Advances in Learning and Control,* V. Blondel, S. Boyd, and H. Kimura, eds, *Lecture Notes in Control and Information Sciences,* Springer (2008) 95–110.

[12] M. Grant and S. Boyd, "CVX: Matlab Software for Disciplined Convex Programming," version 2.0 beta. `http://cvxr.com/cvx` (September 2013).

[13] J. Kallrath, ed., *Modeling Languages in Mathematical Optimization.* Kluwer Academic Publishers, Dordrecht, The Netherlands (2004).

[14] J. Kennedy, "Algorithm To Convert A Decimal To A Fraction." Technical report, Mathematics Department, Santa Monica College (undated); from `sites.google.com/site/johnkennedyshome/home/downloadable-papers/dec2frac.pdf`, uploaded 5 January 2012.

[15] C.A.C. Kuip, "Algebraic Languages for Mathematical Programming." *European Journal of Operational Research* **67** (1993) 25–51.

[16] Miguel Sousa Lobo, Lieven Vandenberghe, Stephen Boyd and Hervé Lebret, "Applications of Second-Order Cone Programming." *Linear Algebra and Its Applications* **284** (1998) 193–228.

[17] J. Löfberg, "YALMIP: A Toolbox for Modeling and Optimization in MATLAB." *Proceedings of the CACSD Conference,* Taipei, Taiwan (2004).

[18] A. Mahajan and T. Munson, Exploiting Second-Order Cone Structure for Global Optimization. Technical report ANL/MCS-P1801-1010, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL (October 28, 2010); from `www.optimization-online.org/DB_HTML/2010/10/2780.html`.