Automatic Reformulation of Second-Order Cone Programming Problems

Victor Zverovich, Robert Fourer

AMPL Optimization

INFORMS Computing Society Conference
January 11-13, 2015, Richmond, Virginia
Second-order cone programming (SOCP)

Problem statement:

\[
\begin{align*}
\text{minimize} & \quad f^T x \\
\text{s.t.} & \quad \|A_i x + b_i\|_2 \leq c_i^T x + d_i, \quad i = 1, \ldots, m \\
& \quad Fx = g,
\end{align*}
\]

where \(x \in \mathbb{R}^n\) is the optimization variable,
\(f \in \mathbb{R}^n, A_i \in \mathbb{R}^{n_i \times n}, b_i \in \mathbb{R}^{n_i}, c_i \in \mathbb{R}^n, d_i \in \mathbb{R}, F \in \mathbb{R}^{p \times n},\) and \(g \in \mathbb{R}^p\).
SOCP constraint

\[ \|Ax + b\|_2 \leq c^T x + d \]
Motivation

Second-order cone programming problems

- Have a wide range of applications:
  - Robust optimization
  - Engineering applications: filter design, antenna array design, etc. See, for example, *Applications of Second-Order Cone Programming* by Lobo et al (1998).
- Can be solved efficiently with interior-point methods
- Many types of problems are convertible to SOCP (Erickson (2013))

But solvers only accept very limited forms of SOCP constraints
Example

minimize $\sqrt{(x + 2)^2 + (y + 1)^2} + \sqrt{(x + y)^2}$
Problem

AMPL model:

```ampl
var x;
var y;
minimize obj: sqrt((x + 2)^2 + (y + 1)^2) + sqrt((x + y)^2);
```

CPLEX:

```ampl
ampl: option solver cplex;
ampl: solve;
CPLEX 12.6.1.0: /tmp/at12668.nl contains a nonlinear objective.
```

MINOS:

```ampl
ampl: option solver minos;
ampl: solve;
MINOS 5.51: Error evaluating objective obj: can't evaluate sqrt'(0).
ampl: let {i in 1.._nvars} _var[i] := 0.1; solve;
MINOS 5.51: optimal solution found?  Optimality tests satisfied, but reduced gradient is large.
12 iterations, objective 2.19544929
Nonlin evals: obj = 126, grad = 125.
```
SOCP reformulation

minimize $u + v$

s. t. $(x + 2)^2 + (y + 1)^2 \leq u^2,$

$(x + y)^2 \leq v^2$

$u, v \geq 0$
Solving SOCP reformulation

AMPL model:

```plaintext
var x;
var y;
var u >= 0;
var v >= 0;
minimize obj: u + v;
s.t. c1: (x + 2)^2 + (y + 1)^2 <= u^2;
s.t. c2: (x + y)^2 <= v^2;
```

CPLEX:

```plaintext
ampl: option solver cplex; solve;
CPLEX 12.6.1.0: QP Hessian is not positive semi-definite.
```

KNITRO:

```plaintext
ampl: option solver knitro; solve;
KNITRO 9.0.1: Locally optimal solution.
objective 2.121313302; feasibility error 9.97e-11
21 iterations; 22 function evaluations
```
Solver forms

"Standard" second-order cone constraint:

\[ \sum_{i=1}^{n} a_i x_i^2 \leq a_{n+1} x_{n+1}^2 \]

where \( a_i \geq 0, x_{n+1} \geq 0 \). Rotated cone constraint:

\[ \sum_{i=1}^{n} a_i x_i^2 \leq a_{n+1} x_{n+1} x_{n+2} \]

where \( a_i \geq 0, x_{n+1} \geq 0, x_{n+2} \geq 0 \).
Making solver happy

AMPL model:

```AMPL
var x;
var y;
var u >= 0;
var v >= 0;
var r;
var s;
var t;
minimize obj: u + v;
s.t. c1: r^2 + s^2 <= u^2;
s.t. c2: t^2 <= v^2;
s.t. c3: x + 2 = r;
s.t. c4: y + 1 = s;
s.t. c5: x + y = t;
```

CPLEX:

```bash
ampl: option solver cplex; solve;
CPLEX 12.6.1.0: optimal solution; objective 2.121320344
5 barrier iterations
```

Works but tedious and error-prone, so...
Let machine do the work
SOCP reformulation system

Features:

- Fast detection of problems convertible to SOCP
- Compatibility with existing solvers: no modifications to the source code of existing solvers required
- Automatic reformulation into SOCP forms accepted by solvers
- Easy to write new transformations
- Modular: components can be reused for different purposes
Architecture

Old

AMPL

AMPL Solver Library

ASL representation

Solver driver

Solver representation

Solver libraries

New

AMPL

AMPL Solver Library

nl reader

New representation

SOCP detector

Not convertible to SOCP

ASL builder (optional)

ASL representation

SOCP converter

Solution

Solver driver

Solver representation

Solver libraries
nl reader

- High performance:
  - mmap-based
  - no dynamic memory allocations
  - handler methods can be inlined
- Simple SAX-like API
- Reusable: not limited to a single problem representation
- Complete
nl reader performance

Text 218.9 MiB

- nl reader
- nl reader+build
- ASL

<table>
<thead>
<tr>
<th>Time, s</th>
<th>nl reader</th>
<th>nl reader+build</th>
<th>ASL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.73</td>
<td>3.74</td>
<td>4.69</td>
</tr>
</tbody>
</table>

Binary 246.9 MiB

<table>
<thead>
<tr>
<th>Time, s</th>
<th>nl reader</th>
<th>nl reader+build</th>
<th>ASL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.50</td>
<td>2.37</td>
<td>3.23</td>
</tr>
</tbody>
</table>

- 730 problems from the CUTE test set
- nl reader w/o problem construction is up to 6x faster than ASL
- Problem construction is faster than ASL, but has room for improvement (pool allocator)
Expression Classes

NumericExpr
- NumericConstant
- Reference (to a variable or a common expression)
- UnaryExpr (unary -, abs, tan, ...)
- BinaryExpr (+, -, *, /, div, less, ...)
- IfExpr
- PLTerm (piecewise-linear term)
- CallExpr
- IteratedExpr (min, max, sum, numberof)
- SymbolicNumberOfExpr
- CountExpr

LogicalExpr
- LogicalConstant (0 or 1)
- NotExpr (!)
- BinaryLogicalExpr (||, &&, <==>)
- RelationalExpr (<, <=, =, !=, >=, >)
- LogicalCountExpr (atleast, atmost, exactly)
- ImplicationExpr (==> else)
- IteratedLogicalExpr (exists, forall)
- PairwiseExpr (alldiff, !alldiff)

(string expressions)
- StringLiteral
- SymbolicIfExpr

Expr
Expression Tree

\[
\text{minimize } \sqrt{(x + 2)^2 + (y + 1)^2 + \sqrt{(x + y)^2}}
\]
Detectors

- Implemented as visitors
- Can be extended via inheritance
- Easy to implement
- Fast: no virtual calls, visit methods can be inlined
- Reusable
Weighted sum detector

Detects arbitrary combinations of sums and multiplications by positive constants:

template <typename Impl>
class WeightedSumDetector : public ExprDetector<Impl> {
    public:
        bool VisitAdd(BinaryExpr e) {
            return this->Visit(e.lhs()) && this->Visit(e.rhs());
        }
        bool VisitMul(BinaryExpr e) {
            return ((IsPosConstant(e.lhs()) && this->Visit(e.rhs())) ||
                    (this->Visit(e.lhs()) && IsPosConstant(e.rhs())));
        }
        bool VisitSum(IteratedExpr e) {
            for (auto arg: e) {
                if (!this->Visit(arg)) return false;
            }
            return true;
        }
};
Affine expression detector

class AffineExprDetector :
    public WeightedSumDetector<AffineExprDetector> {
    public:
        bool VisitNumericConstant(NumericConstant) { return true; }
        bool VisitVariable(Variable) { return true; }
    };
Sum of norms/squares detectors

Detects sums of squares of affine expressions:

```cpp
class SumOfSquaresDetector :
    public WeightedSumDetector<SumOfSquaresDetector> {
    public:
        bool VisitPow2(UnaryExpr e) {
            return AffineExprDetector().Visit(e.arg());
        }
    };
```

Detects sums of norms:

```cpp
class SumOfNormsDetector :
    public WeightedSumDetector<SumOfNormsDetector> {
    public:
        bool VisitSqrt(UnaryExpr e) {
            if (!SumOfSquaresDetector().Visit(e.arg()))
                return false;
            return true;
        }
    };
```
Converters

- Implemented as visitors
- Mirror detectors class hierarchy
- More complicated than detectors, but still easy to implement
- Extensible, fast and reusable
Converters

- **SumConverter<Impl>:** recursively traverses sum (and multiplication by constant) expressions and applies conversion specified by Impl to the terms.
- **AffineExprExtractor:** extracts an affine expression $e$ into a separate constraint $x = e$ where $x$ is a new variable.
- **SumOfSquaresConverter:** replaces each term $ce^2$ with $cx^2$ where $e$ is an affine expression and $x$ is a variable in $x = e$ created by AffineExprExtractor.
- **SumOfNormsConverter:** replaces each term $\sqrt{e}$, where $e$ is a sum of squares, with a new nonnegative variable $x$ and adds a constraint $x^2 \geq e$. 
template <typename Impl>
class SumConverter : public ExprVisitor<Impl, void> {
private:
    Problem &problem_;  
    double coef_; 
public:
    explicit SumConverter(Problem &p) : problem_(p), coef_(1) {}

    void VisitAdd(BinaryExpr e) {
        this->Visit(e.lhs());
        this->Visit(e.rhs());
    }
    void VisitSum(IteratedExpr e) {
        for (auto arg: e)
            this->Visit(arg);
    }
...

Sum of squares converter

class SumOfSquaresConverter :
   public SumConverter<SumOfSquaresConverter> {
private:
   Problem::IteratedExprBuilder &sum_;

public:
   SumOfSquaresConverter(Problem &p, Problem::IteratedExprBuilder &sum) :
      SumConverter<SumOfSquaresConverter>(p), sum_(sum) {}

   void VisitPow2(UnaryExpr e);
};

void SumOfSquaresConverter::VisitPow2(UnaryExpr e) {
   Problem &p = problem();
   AffineExprExtractor extractor(p);
   extractor.Apply(e.arg());
   // Replace the term ``coef * expr ^ 2`` with ``coef * x ^ 2``.
   NumericExpr term = p.MakeUnary(expr::POW2,
      p.MakeVariable(extractor.var_index()));
   if (coef() != 1)
      term = p.MakeBinary(expr::MUL, p.MakeNumericConstant(coef()), term);
   sum_.AddArg(term);
}
SOCP-convertible forms

Quadratic constraints:

\[ \sum_{i=1}^{n} a_i (f_i x + g_i)^2 \leq a_{n+1} (f_{n+1} x + g_{n+1})^2 \]

where \( a_i \geq 0 \) and \( f_{n+1} x + g_{n+1} \geq 0 \) for all feasible \( x \).

\[ \sum_{i=1}^{n} a_i (f_i x + g_i)^2 \leq a_{n+1} (f_{n+1} x + g_{n+1})(f_{n+2} x + g_{n+2}) \]

where \( a_i \geq 0, f_{n+1} x + g_{n+1} \geq 0, f_{n+2} x + g_{n+2} \geq 0 \) for all feasible \( x \).
SOC-representable functions

• A function $\text{SOC}(x)$ is SOC-representable if $\text{SOC}(x) \leq f_{n+1}x + g_{n+1}$ can be equivalently represented by a collection of second-order cone and linear constraints.

• Any positive multiple, sum, or maximum of SOC-representable functions is also SOC-representable.

• Minimization of a SOC-representable function is equivalent to SOCP.
Examples of SOC-representable functions

- \((\sum_{i=1}^{n} a_i |f_i x + g_i|^{\alpha_i})^{1/\alpha_0}\), where \(\alpha_i \geq \alpha_0 \geq 1\). Includes norms.

- Quadratic-linear ratios: \(\frac{\sum_{i=1}^{n} a_i (f_i x + g_i)^2}{f_{n+2} x + g_{n+2}}\)

- Generalization of negative geometric mean: \(\prod_{i=1}^{p} (f_i x + g_i)^{-\alpha_i}\) for rational \(\alpha_i \geq 0\).

and more (see Erickson (2013))
Integration can be a challenge, proper design and planning is important.
Solver integration

- The following ASL functions are replaced using macros
  - `ASL_alloc` - allocates ASL data structure
  - `ASL_free` - frees ASL data structure
  - `jac0dim` - reads an nl file header
  - `qp_read` - reads the rest of an nl file
  - `write_sol` - writes a solution

- No changes to the driver source code, only build config - easy integration

- Can work with any AMPL solver that supports SOCP
Example revisited

AMPL model:

```AMPL
var x;
var y;
minimize obj: sqrt((x + 2)^2 + (y + 1)^2) + sqrt((x + y)^2);
```

CPLEX*

```AMPL
ampl: option solver cplex-socp;
ampl: solve;
CPLEX 12.4.0.0: optimal solution; objective 2.121320344
5 barrier iterations
No basis.
```

It just works!™
Summary

• Current status:
  ▪ Reformulation infrastructure is ready
  ▪ Transformations are being developed

• Future work:
  ▪ More transformations
  ▪ Build solver representation directly: faster, but requires modifications to a solver driver
  ▪ Support more solvers: easy as only recompilation required
References

- Source code: https://github.com/ampl/mp