2013 European Control Conference (ECC)
July 17-19, 2013, Zürich, Switzerland.

Code Generation for Embedded Second-Order Cone Programming

Eric Chu*, Neal Parikh†, Alexander Domahidi‡, and Stephen Boyd*

Abstract— **This paper describes a framework for generating easily verifiable code to solve convex optimization problems in embedded applications by transforming them into equivalent second-order cone programs. In embedded applications, it is critical to be able to verify code correctness, but it is also desirable to be able to rapidly prototype and deploy high-performance solvers for different problems. To balance these two requirements, we propose a code generation system that takes high-level descriptions of convex optimization problems and generates code that maps the parameters in the original problem to data in an equivalent second-order cone program, which is then solved by a single, external solver that can be verified once and for all. A novel aspect is that we restrict the parameters in the original problem to only appear in affine functions, which lets us map the parameters to problem data without performing any floating point operations. As a result, the generated code is lightweight, fast, and trivial to verify. The approach thus marries the benefits of high-level parser/solvers with custom, high-performance, high-reliability solvers for embedded applications.**

I. Introduction

We describe a parser/generator that generates simple, lightweight, and easily verifiable code for solving parameterized families of convex optimization problems on embedded systems. It takes as input a high-level problem description, parses it into an internal representation, verifies the convexity of the problem, and then generates C code that maps the parameters in the original description to the problem data in an equivalent second-order cone program. This resulting code can then be called to solve a particular problem instance, in which the parameters of the original description are fixed to given values. In the generated code, the parameters are mapped to appropriate problem data in a cone program, which is solved with an external solver; the solution of the cone problem is then mapped back to the original variables. We describe each step in detail in what follows.

Many of these ideas are not new: We build on previous work on automatic convexity detection, automatic canonicalization into conic form, and code generation for convex optimization. There are two novel aspects to our system. First, we target second-order cone programs as a canonical form, while previous parser/generators only supported quadratic programs or quadratically-constrained quadratic programs; thus, the scope of problems addressable is much larger here. Second, and most important, we place a greater emphasis on being able to easily verify the correctness of the generated code. In particular, we restrict the use of problem parameters in the problem specification by allowing them to only enter through specific functions. This restriction enables the code generator to map parameters in the original problem to the canonical form without performing *any* floating point operations. The generated code is thus fast, lightweight, trivial to verify, and compatible with embedded hardware providing only fixed point operations.

To solve the resulting second-order cone programs, we call the *embedded conic solver* (ECOS) [1]. (ECOS itself requires floating-point operations.) However, it would be straightforward to use a different solver or to extend our code generation system to also generate the solver code itself. We demonstrate the method on a portfolio optimization problem.

A. Related work

The idea of canonicalizing high-level problem descriptions into cone programs originated with the parser/solver SDPSOL [2]. Other parser/solvers like YALMIP [3] and CVX [4], [5] embed these description languages in Matlab.

While parser/solvers are ideal for prototyping algorithms that rely on convex optimization, deploying these algorithms on embedded systems requires tedious manual transformations and possibly substantial time to write a fast, reliable custom C solver. The parser/generator CVXGEN [6] aims to overcome this limitation by pre-generating the solver for a family of convex optimization problems as custom C source. It is not suitable for rapid prototyping due to the time consuming code generation step, but the resulting solvers are extremely reliable and can solve problem instances on the order of microseconds. However, CVXGEN and similar frameworks, such as FORCES and ACADO, are restricted to solving (a sequence of) quadratic programs or quadratically-constrained quadratic programs [6], [7], [8], [9], [10], [11].

A key ingredient of all these systems is to be able to automatically verify the convexity of a given high-level problem description. Automatic convexity detection appears in systems as diverse as the AMPL modeling language and the solver in Microsoft Excel [12], [13], [14]. Several other approaches exist [15], [16], but in this paper, we use *disciplined convex programming* [4], which we review in §IV. The main advantages are that it consists of a very simple set of rules that come directly from the underlying mathematics; it is easy to implement; and it is easy to canonicalize problems that conform to its rules.

B. Outline

First, in §II, we describe the class of problems we want to solve. In §III, we give a brief overview of different

* E. Chu and S. Boyd are with the Electrical Engineering Department at Stanford University. Email: {echu508, boyd}@stanford.edu
† N. Parikh is with the Computer Science Department at Stanford University. Email: npparikh@cs.stanford.edu
‡ A. Domahidi is with the Automatic Control Laboratory at ETH Zurich. Email: domahidi@control.ee.ethz.ch

approaches to solving these classes of problems. In §IV, we describe the framework of disciplined convex programming and show how it can be used to verify the convexity of the problems discussed in §II. In §V, we describe the canonicalization process, *i.e.*, how we automatically generate an equivalent cone program. We present an example in §VI.

II. Problem families

A parametric family $\mathcal{P}$ of optimization problems is

$$\begin{array}{lll} \text{minimize} & f_0(x;\alpha) & \\ \text{subject to} & f_i(x;\alpha) \le g_i(x;\alpha), & i=1,\dots,m, \\ & u_i(x;\alpha) = v_i(x;\alpha), & i=1,\dots,p, \end{array} \qquad (1)$$

where $x \in \mathbf{R}^n$ is the optimization variable and $\alpha \in \mathcal{A}$ are the parameters. The set $\mathcal{A}$ is called the *parameter space*, and we obtain a particular *problem instance* $\mathcal{P}(\alpha)$ when we choose fixed parameters $\alpha \in \mathcal{A}$. We use the term 'problem' informally, referring either to families or instances; the usage should be clear in context. We assume here that the functions $f_i,\ g_i : \mathbf{R}^n \times \mathcal{A} \to \mathbf{R}$ and $u_i,\ v_i : \mathbf{R}^n \times \mathcal{A} \to \mathbf{R}$ are specified with concrete formulas (described in detail below).

A family $\mathcal{P}$ is convex when, for each $\alpha \in \mathcal{A}$, the functions f_i (including f_0) are convex (in x), the functions g_i are concave, and the functions u_i and v_i are affine. Our focus is on convex families that are *SOCP-representable*, *i.e.*, can be transformed into equivalent second-order cone programs (SOCPs). We focus on these problems because a very large number of important problems are SOCP-representable; see, *e.g.*, [17], [18], [19], [20], [21] for background. SOCP-representable problems include linear programs, quadratic programs, and quadratically constrained quadratic programs.

A convex family of *second-order cone programs* is

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax + s = b, \quad s \in \mathcal{K}, \end{array} \qquad (2)$$

with variable $x \in \mathbf{R}^n$ and $s \in \mathbf{R}^m$. Here $\mathcal{K}$ is the cone

$$\mathcal{K} = \mathcal{Q}^{m_1} \times \cdots \times \mathcal{Q}^{m_q} \times \{0\}^r,$$

where

$$\mathcal{Q}^p = \{(t,x) \in \mathbf{R} \times \mathbf{R}^{p-1} \mid \|x\|_2 \le t\}$$

is the second-order cone of dimension p, and we define $\mathcal{Q}^1$ to be $\mathbf{R}_+$, the cone of nonnegative reals. The parameters in the SOCP are A, b, c, and the cone dimensions $(m_1, \dots, m_q)$. (There are several other standard forms for SOCPs, which are easily transformed into our form, and vice versa.)

We will solve the problem (1) by transforming it into an equivalent SOCP (2), solving the resulting SOCP, and finally, extracting the solution of the original problem from the solution of the SOCP. We refer to the process of transforming a convex problem of the form (1) into an equivalent SOCP of the form (2) as *canonicalization*. By 'equivalent', we mean that we can easily construct a solution of the original problem from a solution of the SOCP. With the canonicalization method described in §V, recovering a solution of the original problem from a solution of the SOCP only involves ignoring the additional variables introduced in the canonicalization.

III. Solvers

There are two main approaches for solving general convex problems: *parser/solvers* and *parser/generators*. A *parser/solver* parses a problem instance $\mathcal{P}(\alpha)$, canonicalizes it to obtain an equivalent SOCP, solves the SOCP (with a generic, external SOCP solver), and then recovers a solution to the original problem from the solution to the SOCP. It is substantially easier to use systems of this type than to use the underlying cone solvers directly, which requires manual transformations of the problem of interest. CVX [22] is an example of a parser/solver.

A *parser/generator* takes a problem *family* $\mathcal{P}$, analyzes it in advance, and generates a custom SOCP solver. When a particular instance $\mathcal{P}(\alpha)$ is to be solved, it only needs to map the parameters in the problem instance to the SOCP problem data, run the (custom) solver, and return a solution to the original problem. Since the analysis of the structure of the problem family can take a substantial amount of time, parser/generators avoid some work when actually solving a particular instance $\mathcal{P}(\alpha)$ and (for small problems) can solve individual instances orders of magnitude faster than other approaches. CVXGEN [6] is an example of a parser/generator.

Our approach takes a problem family $\mathcal{P}$, canonicalizes it, and generates lightweight code for mapping parameters of problem instances $\mathcal{P}(\alpha)$ into SOCP problem data. In particular, our mapping code, called `prob2socp`, performs no floating point operations; it merely copies data into the appropriate locations in the SOCP problem data. The converse operation, called `socp2prob`, takes a solution of the SOCP and returns a solution of the original problem. It, too, does nothing more than copy data from one structure to another. To solve instances of the original problem, we simply call `prob2socp`, the SOCP solver (ECOS, in our case), and `socp2prob` in order. If many problems from the same family are to be solved, these three functions are called repeatedly. Setup routines and tasks such as determining the elimination order in ECOS or allocating memory can be done beforehand, and only once.

IV. Disciplined convex programming

In this section, we describe how convex problem families and problem instances are represented, and how we can verify convexity of the problem family or instance.

A. Atoms

The functions f_i, g_i, u_i, and v_i in the problem instance or family are described using a set of given *atoms*, which are built-in functions such as sum, maximum, square, or norm. A full list of the atoms is given in Table I.

Our system supports *vector*, *scalar*, and *parametric* atoms. A vector atom takes vector arguments and returns a scalar, while a scalar atom takes scalar arguments and returns a scalar. When scalar atoms are evaluated with vector arguments, they are applied elementwise. A parametric atom has (vector or matrix) parameters in addition to arguments.

Each atom has three more key properties: the *sign* of its output (positive, negative, or unknown), its *monotonicity* in each argument (increasing, decreasing, or neither), and its *curvature* (convex, concave, or affine). For example, the atom ϕ^{sqrt} (the square root) is positive, increasing, and concave. (We use positive to denote nonnegative, increasing to mean nondecreasing, and so on.)

More generally, atoms can have *sign-dependent sign* and *sign-dependent monotonicity*. For example, the atom ϕ^{plus} (which adds its two scalar arguments) is positive when both arguments are positive, negative when both arguments are negative, and neither otherwise. As another example, ϕ^{square} (the square) is increasing when its argument is positive and decreasing when its argument is negative. (Though not currently supported, this idea could be extended to support *sign-dependent curvature*; *e.g.*, a 'cube' atom ϕ^{cube} would be convex (concave) for positive (negative) arguments.)

The sign and monotonicity of parametric atoms depends on the sign of the associated parameters, which in turn depends on $\mathcal{A}$. For example, ϕ^{smult} (scalar multiplication) is positive when its parameter α and argument x are positive; it is increasing when its parameter α is positive. (All parametric atoms are affine, so the curvature does not depend on the sign of the parameter or argument.)

A vector is considered to be positive only if every entry is positive. For example, the atom

$$\phi^{\text{quad\_over\_lin}}(x, y) = \frac{x^T x}{y},$$

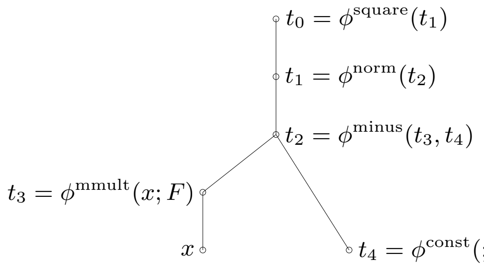
where x is a vector and $y > 0$ is a scalar, is positive, nonmonotonic in x, decreasing in y, and convex. When x is positive (negative), it is increasing (decreasing) in x.

B. Expressions

An *expression* is recursively defined as either a (vector) variable or an atom evaluated at a subexpression. They are naturally represented as trees in which the leaves are variables, the internal nodes are atoms, and each atom is considered to be evaluated using its children (which may be variables or more complex subexpressions) as arguments. See Figure 1 for an example. We distinguish between atoms and expressions: For example, ϕ^{square} is an atom, while $\phi^{\text{square}}(x)$ is an expression (assuming x is a variable).

We can determine the sign of an expression in the following recursive manner. Variables have unknown sign. The sign of an expression is then determined by working our way up the expression tree and using the sign rules for the atoms at each internal node. For example, $\phi^{\text{square}}(x) + \phi^{\text{square}}(y)$ is positive because the sign of ϕ^{square} is positive even when evaluated with an expression of unknown sign, and the sign of ϕ^{plus} (which we write as $+$ in infix notation above) is positive when both its arguments are positive.

We can build on this to determine the curvature of an expression in the following recursive fashion. Variables are affine. We then work up the expression tree applying the following composition rule: $\phi(g_1(x), \ldots, g_n(x); \alpha)$ is convex if the atom ϕ is convex and if, for each i, we have that either g_i is affine, ϕ is increasing in the ith argument and g_i is convex, or ϕ is decreasing in the ith argument and g_i



$t_0 = \phi^{\text{square}}(t_1)$
$t_1 = \phi^{\text{norm}}(t_2)$
$t_2 = \phi^{\text{minus}}(t_3, t_4)$
$t_3 = \phi^{\text{mmult}}(x; F)$
x
$t_4 = \phi^{\text{const}}(; g)$

Fig. 1: Expression tree representation of $\|Fx - g\|_2^2$.

is concave. The rule for verifying concavity is similar. The expression is affine if ϕ is affine and $g_1, \ldots, g_n$ are all affine.

C. Convex problems

We can verify the convexity of a problem family by determining the curvature of all the expressions and atoms that appear in it. In particular, a problem family is convex when, for each $\alpha \in \mathcal{A}$, the following three conditions hold:

1) the objective function f_0 is convex;
2) the functions f_i (g_i) are convex (concave);
3) the functions u_i and v_i are affine.

We refer to problems that are convex when verified this way as *DCP-compliant*. All DCP-compliant problems are convex, but the converse is not true; there are convex problems that cannot be verified as convex using the recursive rules described above.

V. Canonicalization

Given a DCP-compliant problem family $\mathcal{P}$, the next step is canonicalization: automatically transforming this problem into an equivalent second-order cone program. This process is carried out (at least conceptually) in several stages.

A. Smith form

The first step is to rewrite the problem in *Smith form* [23], [24], which involves introducing a new variable for each subexpression. This is best illustrated with an example. Consider the least-squares problem

$$\text{minimize} \quad f(x) = \|Fx - g\|_2^2$$

with $x \in \mathbf{R}^n$ and problem data $F \in \mathbf{R}^{m \times n}$ and $g \in \mathbf{R}^m$. Using the atoms in Table I, we express f as

$$f(x) = \phi^{\text{square}}(\phi^{\text{norm}}(\phi^{\text{minus}}(\phi^{\text{mmult}}(x; F), \phi^{\text{const}}(; g)))).$$

The Smith form of this problem is then the following:

$$\begin{array}{ll} \text{minimize} & t_0 \\ \text{subject to} & t_0 = \phi^{\text{square}}(t_1) \\ & t_1 = \phi^{\text{norm}}(t_2) \\ & t_2 = \phi^{\text{minus}}(t_3, t_4) \\ & t_3 = \phi^{\text{mmult}}(x; F) \\ & t_4 = \phi^{\text{const}}(; g), \end{array}$$

where the variables are (the original one) x and (new ones) $t_0 \in \mathbf{R}$, $t_1 \in \mathbf{R}$, $t_2 \in \mathbf{R}^m$ $t_3 \in \mathbf{R}^m$, and $t_4 \in \mathbf{R}^m$. Each of the new variables corresponds to a node (atom) in the expression tree for the original objective $\|Fx - g\|_2^2$, and each atom's

TABLE I: Atom library.

Atom	Definition	Curvature	Implementation
Parametric atoms			
$\phi^{\text{smult}}(x;a)$	ax	Affine	$t=ax$
$\phi^{\text{mmult}}(x;A)$	Ax	Affine	$t=Ax$
$\phi^{\text{const}}(;a)$	a	Affine	$t=a$
Scalar atoms			
$\phi^{\text{plus}}(x,y)$	$x+y$	Affine	$t=x+y$
$\phi^{\text{minus}}(x,y)$	$x-y$	Affine	$t=x-y$
$\phi^{\text{negate}}(x)$	$-x$	Affine	$t=-x$
$\phi^{\text{pos}}(x)$	$\max(x,0)$	Convex	$t\in\mathcal{Q}^1,\ t-x\in\mathcal{Q}^1$
$\phi^{\text{neg}}(x)$	$\max(-x,0)$	Convex	$t\in\mathcal{Q}^1,\ t+x\in\mathcal{Q}^1$
$\phi^{\text{square}}(x)$	x^2	Convex	$t\geq\phi^{\text{quad\_over\_lin}}(x,1)$
$\phi^{\text{inv\_pos}}(x)$	$1/x$	Convex	$t\geq\phi^{\text{quad\_over\_lin}}(1,x)$
$\phi^{\text{abs}}(x)$	$\lvert x\rvert$	Convex	$(t,x)\in\mathcal{Q}^2$
$\phi^{\text{geo\_mean}}(x,y)$	$\sqrt{xy}$	Concave	$((1/2)(y+x),(1/2)(y-x),t)\in\mathcal{Q}^3,\ y\in\mathcal{Q}^1$
$\phi^{\text{sqrt}}(x)$	$\sqrt{x}$	Concave	$t\leq\phi^{\text{geo\_mean}}(x,1)$
Vector atoms			
$\phi^{\text{sum}}(x)$	$\mathbf{1}^Tx$	Affine	$t=\mathbf{1}^Tx$
$\phi^{\text{max}}(x)$	$\max\{x_1,x_2,\ldots,x_n\}$	Convex	$t-x_i\in\mathcal{Q}^1,\ i=1,\ldots,n$
$\phi^{\text{quad\_over\_lin}}(x,y)$	x^Tx/y	Convex	$((1/2)(y+t),(1/2)(y-t),x)\in\mathcal{Q}^{n+2},\ y\in\mathcal{Q}^1$
$\phi^{\text{norm}}(x)$	$\lVert x\rVert_2$	Convex	$(t,x)\in\mathcal{Q}^{n+1}$
$\phi^{\text{norm1}}(x)$	$\lVert x\rVert_1$	Convex	$t\geq\phi^{\text{sum}}(\phi^{\text{abs}}(x))$
$\phi^{\text{norm\_inf}}(x)$	$\lVert x\rVert_\infty$	Convex	$t\geq\phi^{\text{max}}(\phi^{\text{abs}}(x))$
$\phi^{\text{min}}(x)$	$\min\{x_1,x_2,\ldots,x_n\}$	Concave	$x_i-t\in\mathcal{Q}^1,\ i=1,\ldots,n$

arguments are now individual variables. Furthermore, each constraint involves only a *single* atom.

Figure 1 shows the expression tree representation (and the new variables introduced) for this problem. Parameters are private to the ϕ^{mmult} and ϕ^{constant} atoms and are not represented in the tree itself.

B. Relaxed Smith form

The Smith form is of course equivalent to the original problem, but it is not convex (unless all atoms used are affine), since in a convex problem, all equality constraints must be affine. The next step is to *relax* the equality constraints with nonlinear atoms into inequality constraints as follows. If an atom ϕ is convex, the constraint $t=\phi(x)$ is replaced with $t\geq\phi(x)$, and if ϕ is concave, it is replaced with $t\leq\phi(x)$. If ϕ is affine, the constraint remains as is.

We refer to the resulting problem as being in *relaxed Smith form*. When the original problem is DCP-compliant, the relaxation is guaranteed to be *tight* in the sense that if $(x^\star,t^\star)$ is optimal for the relaxed Smith form, then $x^\star$ is optimal for the original problem. Here, $x^\star$ refers to all the original variables, and $t^\star$ refers to all the new variables introduced in Smith form.

The relaxed Smith form for our least-squares example is

$$\begin{array}{ll} \text{minimize} & t_0 \\ \text{subject to} & t_0\geq\phi^{\text{square}}(t_1) \\ & t_1\geq\phi^{\text{norm}}(t_2) \\ & t_2=\phi^{\text{minus}}(t_3,t_4) \\ & t_3=\phi^{\text{mmult}}(x;F) \\ & t_4=\phi^{\text{const}}(;g). \end{array}$$

This is a convex problem, equivalent to the original least-squares problem. (For this example, it is easy to see that for any solution of the relaxed Smith form problem, the two inequality constraints hold as equalities.)

C. Representation of atoms

The next question concerns how atoms are represented. We use *graph implementations* [5] to represent nonlinear functions in our atom library. (Affine atoms do not require any special representation.) At a high level, each nonlinear atom is defined as the optimal value of a partially-specified convex optimization problem. A convex atom is defined via a minimization problem; a concave atom is defined via a maximization problem.

For example, the atom ϕ^{abs} is represented as

$$\begin{array}{ll} \text{minimize} & t \\ \text{subject to} & (t,x)\in\mathcal{Q}^2. \end{array}$$

The optimal value of this problem for a given value of x is precisely $|x|$. Whenever we see the atom ϕ^{abs} in the relaxed Smith form of a problem, we incorporate this representation into the surrounding problem. As atoms only appear individually in the constraints when problems are written in relaxed Smith form, we need only define how the constraint $t\geq\phi(x)$ $\big(t\leq\phi(x)\big)$ is rewritten for convex (concave) atoms. For example, $t\geq\phi^{\text{abs}}(x)$ would be replaced with $(t,x)\in\mathcal{Q}^2$. (Care is taken to avoid name conflicts with existing variables.) Table I gives a listing of the atoms, their definitions, and their equivalent conic implementations (*i.e.*, the inline expansion of the relaxed Smith form constraint).

When each atom in the relaxed Smith form is replaced by its graph implementation (each of which is a small SOCP), we obtain an SOCP, and the canonicalization is complete. Graph implementations can also depend on other atoms; for example, the implementation of ϕ^{sqrt} is given by maximizing a variable t subject to $t \le \phi^{\text{geo\_mean}}(x, 1)$. In such cases, graph implementations must be unrolled recursively.

Expanding the two nonlinear atoms ϕ^{square} and ϕ^{norm} that appear in our least-squares example, we arrive at the SOCP

$$\begin{array}{ll} \text{minimize} & t_0 \\ \text{subject to} & (1+t_0, 1-t_0, 2t_1) \in \mathcal{Q}^3 \\ & (t_1, t_2) \in \mathcal{Q}^{m+1} \\ & t_2 = t_3 - t_4 \\ & t_3 = Fx \\ & t_4 = g, \end{array}$$

where we write the affine atoms in their natural form.

The final step is to express this SOCP in the standard form (2). With variable $(x, t_0, \ldots, t_4)$, and the same order of equations as above, the SOCP problem data are

$$c = (0, 1, 0, \ldots, 0),$$

$$A = \left[\begin{array}{cccccc} & -1 & & & & \\ & 1 & & & & \\ & & -2 & & & \\ \hline & & -1 & & & \\ & & & -I & & \\ \hline & & & I & -I & I \\ F & & & & -I & \\ & & & & & I \end{array}\right], \quad b = \left[\begin{array}{c} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ g \end{array}\right]$$

(with empty entries meaning zero and the horizontal lines giving the cone boundaries), and cone dimensions

$$m_1 = 3, \quad m_2 = m + 1.$$

The first three rows of A, b correspond to the ϕ^{square} atom; this pattern is fixed and repeated whenever the ϕ^{square} atom is used. Similarly, for other atoms, their SOCP definition is effectively *copied* into the appropriate block of A, b.

Note that, as a consequence of Smith form, we introduced new variables (and thus new columns in A) for *all* affine atoms; this is not strictly necessary since affine expressions can be handled explicitly. These variables affect solver performance and can be eliminated via a pre-solver.

D. Mapping parameters to problem data

Examining the canonicalization procedure, we see that the original problem parameters only appear in affine atoms. Moreover, the original problem parameters are only copied into either the coefficient matrix or the righthand side of the data for the SOCP. Thus, the canonicalization procedure is very simple: it consists of copying the problem data into the SOCP data. Both of these data can be represented as C `structs`, so the canonicalization consists of nothing more than copying entries from the `struct` that holds the problem data into the `struct` that holds the SOCP data. If the parameter data provides a `copy` interface, then we can handle more generic data objects. All of the other entries in the SOCP data derive from atom definitions and are fixed, *i.e.*, do not depend on the problem data.

Our simple example makes this mapping very clear. The function `prob2socp` constructs the SOCP data, which involves copying F into the matrix A, and g into the vector b; the other entries in the SOCP data are constant. The function `socp2prob` is even simpler: it simply takes the first m components from the SOCP solution vector.

VI. Portfolio optimization example

We consider a simple long-only portfolio optimization problem [17, p. 185–186], where we choose relative weights of assets to maximize risk-adjusted return. The problem is

$$\begin{array}{ll} \text{maximize} & \mu^T x - \gamma(x^T \Sigma x) \\ \text{subject to} & \mathbf{1}^T x = 1, \quad x \ge 0, \end{array}$$

where the variable $x \in \mathbf{R}^n$ represents the portfolio, $\mu \in \mathbf{R}^n$ is the vector of expected returns, $\gamma > 0$ is the risk-aversion parameter, and $\Sigma \in \mathbf{R}^{n \times n}$ is the asset return covariance, given in factor model form,

$$\Sigma = FF^T + D.$$

Here $F \in \mathbf{R}^{n \times m}$ is the factor-loading matrix, and $D \in \mathbf{R}^{n \times n}$ is a diagonal matrix (of *idiosyncratic risk*). The number of factors in the risk model is m, which we assume is substantially smaller than n, the number of assets.

Using the structure of Σ, we can express the problem as

$$\begin{array}{ll} \text{maximize} & \mu^T x - \gamma \|F^T x\|_2^2 - \gamma \|D^{1/2} x\|_2^2 \\ \text{subject to} & \mathbf{1}^T x = 1, \quad x \ge 0. \end{array}$$

In our domain-specific language, this problem is:

```
variable x(n)
parameter mu(n)
parameter gamma positive
parameter F(n,m)
parameter Dhalf(n,n)

maximize mu'*x
    - gamma*square(norm(F'*x))
    - gamma*square(norm(Dhalf*x))
subject to
  sum(x) == 1
  x >= 0
```

From this specification, our code generator creates `prob2socp` and `socp2prob`. Structure in the data (*e.g.*, the fact that `Dhalf` is diagonal) is automatically exploited.

This code snippet also illustrates some language features:

- *Infix notation.* The multiplication operator, expressed with $*$ in infix notation, is the parametric atom $\phi^{\text{mult}}(x; A) = Ax$. The subtraction operator, expressed with $-$ in infix notation, is the scalar atom ϕ^{minus}.
- *Vector atoms.* The `sum` function is the vector atom ϕ^{sum}, which takes a vector and returns a scalar.
- *Signed monotonicity.* `square(norm())` is correctly recognized as convex since `norm` is positive and `square` is increasing when its argument is positive.
- *Constants and scalar-vector promotion.* The (scalar) constant `0` is treated as a scalar parameter; it is then

TABLE II: Summary for portfolio example.

(m, n)	CVX	ECOS
$(10, 300)$	173ms	18ms
$(20, 500)$	215ms	57ms
$(30, 1000)$	324ms	311ms

promoted to a vector and the inequality is interpreted elementwise.

- *Quadratic forms.* Because we require parameters to only appear in affine functions, we do not have an atom to define quadratic forms, as in

$$\phi^{\text{quad\_form}}(x; P) = x^T P x.$$

To express quadratic forms, the user precomputes the Cholesky factor or square root of P and writes

$$\phi^{\text{square}}(\phi^{\text{norm}}(P^{1/2}x)) = \|P^{1/2}x\|_2^2.$$

This was done for $x^T D x$ in our example.

On a 3.4GHz Intel Xeon machine with 16GB of RAM, we compare our solve times for three small problems to CVX (using SeDuMi as the underlying solver). The results are summarized in Table II. Note that SeDuMi is multithreaded. The ECOS times are approximately 3 times slower than the times in [1]; this is due to the lack of a pre-solver and the introduction of new (redundant) variables in the relaxed Smith form. A pre-solver recovers the performance in [1].

VII. Conclusion

In this paper, we have described a framework for generating easily verifiable code to solve SOCP-representable problem families $\mathcal{P}$ on embedded systems. The key is to restrict parameters to only appear in affine, parametric atoms.

A high-level description language for the problem is translated into a lightweight wrapper that

1) copies problem instance $\mathcal{P}(\alpha)$'s parameters α into SOCP data A, b, and c,
2) calls the ECOS solver, and
3) copies the solution back into the original variables.

Because our generated code only copies memory, it is easy to verify, and it is compatible with fixed-point operations. It relies on a fast and robust SOCP solver implementation, ECOS. By validating ECOS, users may adapt problem families as needed for embedded applications without needing to validate the generated code whenever the problem changes. Thus, we are able to marry the benefits of high-level parser/solvers with custom, high-performance, high-reliability solvers for embedded applications.

Acknowledgements

This work was supported by DARPA XDATA (FA8750-12-2-0306), NASA (NNX07AEIIA), and EMBOCON (ICT-248940) from the EU in FP7. N. Parikh was also supported by a NSF Graduate Research Fellowship (DGE-0645962).

References

[1] A. Domahidi, E. Chu, and S. Boyd, "ECOS: An SOCP solver for embedded systems," in *European Control Converence*, Zurich, Switzerland, 2013, to appear.

[2] S.-P. Wu and S. Boyd, "SDPSOL: A parser/solver for SDP and MAXDET problems with matrix structure," Available at www.stanford.edu/~boyd/old_software/SDPSOL.html, Nov 1995.

[3] J. Lofberg, "YALMIP: A toolbox for modeling and optimization in MATLAB," in *IEEE International Symposium on Computed Aided Control Systems Design*, Sep 2004, pp. 294–289.

[4] M. Grant, S. Boyd, and Y. Ye, "Disciplined convex programming," in *Global Optimization: From Theory to Implementation*, ser. Nonconvex Optimization and its Applications, L. Liberti and N. Maculan, Eds. Springer, 2006, pp. 155–210.

[5] M. Grant and S. Boyd, "Graph implementations for nonsmooth convex programs," in *Recent Advances in Learning and Control*, ser. Lecture Notes in Control and Information Sciences, V. Blondel, S. Boyd, and H. Kimura, Eds. Springer, 2008, pp. 95–110.

[6] J. Mattingley and S. Boyd, "CVXGEN: A code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, pp. 1–27, 2012.

[7] A. Domahidi, A. Zgraggen, M. Zeilinger, M. Morari, and C. Jones, "Efficient interior point methods for multistage problems arising in receding horizon control," in *IEEE Conference on Decision and Control*, Grand Wailea Maui, HI, USA, Dec. 2012.

[8] A. Domahidi, "FORCES: Fast optimization for real-time control on embedded systems," Available at forces.ethz.ch/, Oct. 2012.

[9] B. Houska, H. Ferreau, and M. Diehl, "ACADO Toolkit – An open source framework for automatic control and dynamic optimization," *Optimal Control Applications and Methods*, vol. 32, no. 3, pp. 298–312, 2011.

[10] B. Houska and H. Ferreau, "ACADO toolkit user's manual," www.acadotoolkit.org, 2009–2011.

[11] D. Ariens, B. Houska, and H. Ferreau, "Acado for Matlab user's manual," www.acadotoolkit.org, 2010–2011.

[12] Frontline Solvers, "Excel solver, optimization software, Monte Carlo simulation, data mining - Frontline Systems," www.solver.com.

[13] R. Fourer, C. Maheshwari, A. Neumaier, D. Orban, and H. Schicl, "Convexity and concavity detection in computation graphs: Tree walks for convexity assessment," *INFORMS Journal on Computing*, vol. 22, pp. 26–43, 2010.

[14] I. P. Nenov, D. H. Fylstra, and L. V. Kolev, "Convexity determination in the Microsoft Excel solver using automatic differentiation techniques," in *Fourth International Workshop on Automatic Differentiation*, 2004.

[15] D. R. Stoutmeyer, "Automatic categorization of optimization problems: An application of computer symbolic mathematics," *Operations Research*, vol. 26, pp. 773–738, 1978.

[16] C. Crusius, "Automated analysis of convexity properties of nonlinear programs," Ph.D. dissertation, Stanford University, 2003.

[17] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

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

[19] Y. Nesterov and A. Nemirovskii, *Interior-Point Polynomial Methods in Convex Programming*. Society for Industrial and Applied Mathematics, 1994.

[20] F. Alizadeh and D. Goldfarb, "Second-order cone programming," *Mathematical Programming Series B*, vol. 95, pp. 3–51, 2003.

[21] M. S. Lobo, L. Vandenberghe, S. Boyd, and H. Lebret, "Applications of second-order cone programming," *Linear Algebra and Its Applications*, vol. 284, pp. 193–228, 1998.

[22] M. Grant, S. Boyd, and Y. Ye, "CVX: Matlab software for disciplined convex programming, ver. 2.0, build 870," Available at www.stanford.edu/~boyd/cvx/, Sep. 2012.

[23] E. Smith, "On the optimal design of continuous processes." Ph.D. dissertation, Imperial College London (University of London), 1996.

[24] L. S. Liberti, "Reformulation and convex relaxation techniques for global optimization," Ph.D. dissertation, Imperial College London (University of London), 2004.