CVXcanon: Automatic Canonicalization of Disciplined Convex Programs

John Miller Paul Quigley Jack Zhu

June 5, 2015

Abstract

A number convex optimization modeling tools have been written in high level languages including MATLAB, Python, and Julia [MUB] [SDB14] [GB14]. These tools translate high-level problem descriptions into low-level, canonical forms that are then passed to an appropriate solver. In this project, we develop CVXcanon, a software package that factors out the common operations all such modeling systems perform into a single library with a simple C++ interface. CVXcanon is currently interfaced with CVXPY and provides 2-10x speedups over the pure Python implementation.

1 Introduction

Modeling languages for convex optimization translate problems from user-friendly, high-level languages into solver-friendly representations. The strength of this approach is that problems can be specified in their natural mathematical form, and the transformation is carried out automatically by software.

Despite the universality of the final representations, much of the logic for carrying out problem transformations is duplicated across CVX, CVXPY, and Convex.JL, collectively refered to as CVX.* [MUB] [SDB14] [GB14].This state of affairs slows the development of modeling tools in new programming languages and needlessly limits improvements to the transformation process to a particular modeling system.

CVXcanon addresses this problem by unifying the *canonicalization* process performed by each of the CVX.* systems in a single optimized, low-level library. The canonicalization procedure typically dominates the processing time in CVX.* system. CVXcanon substantially reduces the time required to process many problems and can be viewed as a first step towards unifying much of the functionality of CVX.* in a single low-level library.

2 Canonicalization

All of the modeling systems in the CVX.* family carry out the transformation from problem specification to standard form in a similar fashion, depicted graphically in Figure 1 and

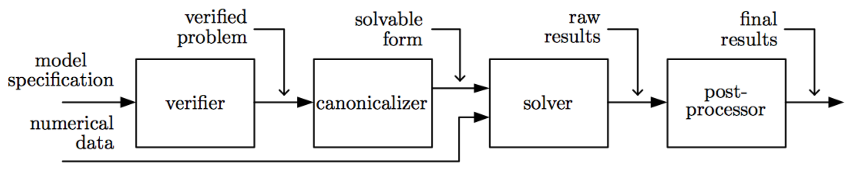

Figure 1: A basic outline of the DCP workflow, taken from [Gra04].

described in detail in [Gra04] and [MUB].

CVXcanon targets the canonicalization step. During canonicalization, the abstract syntax tree describing the problem is converted into a standard matrix form for use by a backend solver. We first describe the standard form used by solvers supported by CVX.* and then explicitly describe the transformation process.

2.1 Cone Programs

A cone program is a convex optimization problem of the form

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \in \mathcal{K}, \end{array} \tag{1}$$

where $x \in \mathbf{R}^n$ is the optimization variable, $A \in \mathbf{R}^{m \times n}$, $b \in \mathbf{R}^m$ and $c \in \mathbf{R}^n$ are problem data, and $\mathcal{K}$ is a convex cone. Typically, $\mathcal{K}$ is given as the Cartesian product of several simpler cones,

$$\mathcal{K} = \mathcal{K}_1 \times \mathcal{K}_2 \times \cdots \times \mathcal{K}_p,$$

where each cone $\mathcal{K}_i$ is from the following list:

- Zero Cone: $\mathcal{K}_0 = \{0\}$.
- Free Cone: $\mathcal{K}_{\text{free}} = \mathbf{R}$
- Non-negative Cone: $\mathcal{K}_+^n = \{x \in \mathbf{R}^n : x \succeq 0\}$
- Second Order Cone: $\mathcal{K}_{\text{SOC}}^n = \left\{(x,t) \in \mathbf{R}^{n+1} : \|x\|_2 \leq t\right\}$.
- Positive Semidefinite Cone: $\mathcal{K}_{\text{PSD}}^n = \left\{\mathbf{vec}(X) : X \in \mathbf{S}^n \; z^T X z \geq 0 \text{ for all } z \in \mathbf{R}^n\right\}$, where $\mathbf{vec}(X) \in \mathbf{R}^{n^2}$ is the vectorized version of X.

The problem is specified by the matrix A, vectors b and c, and the cone $\mathcal{K}$. The goal of canonicalization is to take as input an arbitrary disciplined convex program and construct an equivalent program in standard form. This process requires care to change all of the expressions in a program to standard form. For example, CVXPY contains a convolution atom, `conv`, which must be converted to dense matrix multiplication with an appropriate Toeplitz matrix.

2.2 Linearization

In general, the canonicalization procedure must reformulate the problem so that all of the expressions are affine and all of the constraints are cone constraints. This transformation is always possible because the CVX.* family of modelling languages require the input program to satisfy the disciplined convex programming ruleset [Gra04]. Adherence to the ruleset ensures that each of the functions appearing the problem is *cone-representable*.

A function is cone-representable if the value of the function is the optimal value of some conic form optimization problem. For example, the absolute value function $f(x) = |x|$ is cone-representable because

$$|x| = \begin{array}{ll} \text{minimize} & t \\ \text{subject to} & (t,x) \in \mathcal{K}_{\text{SOC}}, \end{array} \tag{2}$$

where $t \in \mathbf{R}$ is the optimization variable and $x \in \mathbf{R}$ is the problem data. Such a representation is called the *graph form* of the function [GB08]. In general, all of the atoms in the CVX.* systems implement a function that retuns the graph form of the atom.

Concretely, the graph form of an atom is represented as a tuple (s, o, C). In this representation, the element s is the sense of the objective, the element o is an abstract syntax tree representing the linear objective, and the element C is a set of constraints where each element $c_i \in C$ is itself a tuple $(e_i, \mathcal{K}_i)$ representing a constraint $g(x) \in \mathcal{K}_i$ and e_i is an abstract syntax tree representing $g(x)$. Standard cone form requires all of the expressions to be affine, so o and e_i contain only linear atoms.

Expressions in CVX.* are themselves represented as abstract syntax trees composed of several atoms. The cone representation of the expression can be computed via a simple recursive algorithm given in [MUB]. Let e be root of the abstract syntax tree.

- Base Case: If e is affine, return.
- For each child atom of e, compute the graph implementation: (s_i, o_i, C_i).
- Recurse on each of the expressions in the objectives $\{o_1, \ldots, o_n\}$ to obtain the top-level problem (s, o, C).
- Concatenate the lists of constraints and return $(s, o, C \cup_i C_i)$.

The output of the algorithm is an equivalent expression tree containing only linear atoms. All of the non-linearities in the original expression are captured in the cone constraints. We refer to the resulting expression tree as a *linear expression tree*.

2.3 Matrix Representation

The final step in the canonicalization procedure is to convert the linear expression trees into standard form for backend solvers, including fully specifying the problem data A, b, and c, as well as the cone $\mathcal{K}$. Mathematically, this procedure is straight-forward. Each linear atom is represented as a matrix-vector product. For example, $\phi^{\text{sum}}(x)$ is represented as $\mathbf{1}^T x$. The representation of an expression is the composition of the matrices for each linear atom along a path in the abstract syntax tree. To construct the problem data, one simply concatenates all of the variables into a single vector, partitions A, b, and c into blocks for each subset of variables, and then adds the matrix-vector representation of the corresponding expression to A, b, and c.

3 CVXcanon

Although conceptually simple, the final stage of canonicalization, matrix representation, is by far the most computationally intensive and often dominates the time required by CVX.* systems to process a problem. For many problems, this conversion step actually takes longer than the time to solve the problem. CVXcanon is a C++ library that specifically targets the matrix representation process for optimization in a low-level, compiled language. In particular, CVXcanon converts a problem from its linear expression tree representation to its final cone representation.

Matrix representation algorithm CVXcanon abstracts away the details of the underlying matrix representation algorithm from the high-level language implementation, allowing CVX.* systems to simply pass a list of linear expression trees to the `get_problem_matrix` function and receive a standard conic form representation of the problem.

Internally, CVXcanon traverses each linear expression tree, determines the coefficients corresponding to each atom, multiplies the atoms descendants by these coefficients, then places the resultant matrices into the problem matrix. A detailed description of the matrix stuffing algorithm is given in the Appendix A.

One technical challenge that arises during matrix construction is that the data matrix A is extremely sparse, even when the original problem data is dense. Therefore, both the internal CVXcanon matrix representation, as well as the representation of the matrix returned to CVX.* must also be sparse lest the system incur unacceptable performance penalties.

Wrappers. Our C++ code interacts with high-level languages through SWIG and C bindings. Future implementations of CVX in other languages can utilize these generalized bindings to avoiding rewriting algorithms. Currently, CVXcanon only has a Python wrapper. A natural next step is to expand the wrappers to include other languages, *e.g.* Julia and Matlab. In Python, CVXcanon can be called in a single line:

```
import canonInterface
V, I, J, b = canonInterface.get_problem_matrix(lin_expr_trees).
```

Here, `lin_expr_trees` is a list of CVXPY linear expression trees, V, I, and J is a coordinate list (COO) representation of the data matrix A, and b is a dense data vector.

Codebase All of our code is open-source. A repository containing the CVXcanon codebase, documentation, and installation instructions is on Github:

`http://github.com/jacklzhu/CVXcanon`

A fork of CVXPY modifed to use CVXcanon for canonicalization is also on Github:

`http://github.com/jacklzhu/CVXPY`

4 Performance

CVXcanon significantly decreases the end-to-end solve time of CVXPY for a large range of problems. Figure 2 shows the speedup factor, as defined by the ratio of the baseline CVXPY solve-time to the modifed CVXPY with CVXcanon solve-time, on a variety of EE364A homework problems [BV04]. Performance was tested on a Dell XPS 13 Developer Edition Laptop running Ubuntu 12.04, with an Intel Core i5-4200 CPU @ 1.60 GHz × 4 and 7.3 GB RAM. Each script was run 30 times and the medians are plotted with error bars at the quartiles.

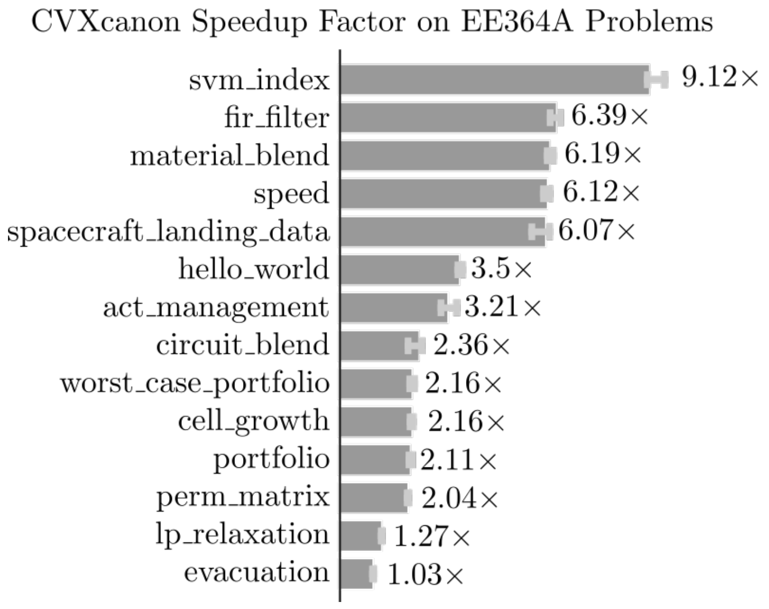

Figure 2: Speed Comparison of baseline CVXPY versus CVXPY backed with CVXCanon

While we observe speedups for all problems tested, this magnitude of the speed increase is largely dependent on the problem description. In some cases, like `lp_relaxation`, the problem is presented in essentially canonical form. In these cases, the canonicalization step occurs almost instantly, and CVXcanon does not offer performance gains over the baseline implementation. In other cases, the problem description results in extremely large linear expression trees, which incur large loop overheads in Python during canonicalization. This is especially true if the problem is not presented in vectorized form. For example, the `svm_index` problem description contains a very large number of vector indexing atoms. In this case, CVXcanon's compiled implementation avoids the loop overhead cost and obtains a speedup of 9 times over the baseline implemenation.

For a substantial subset of problems, the time required to canonicalize the problem in CVXPY exceeds the time required to actually solve the problem. In the table below, we compare the canonicalzation time of CVXcanon and the canonicalzation time of baseline CVXPY on a number of such problems. Code for each problem is presented in Appendix B.

	CVXPY Canonicalization Time	CVXcanon Canonicalization Time
Summation	6.91 s	563 ms
Indexing	17.4 s	7.93 s
Transpose	464 ms	210 ms
Matrix Constraint	318 ms	167 ms
Matrix Product	5.77 s	1.96 s
SVM with Index	6.03 s	526 ms

CVXcanon performs extremely well on all of these tasks, providing from 2x-10x performance improvements across the board. The speedup is likely accounted for by avoiding substantial loop overheads by using a compiled langauge and C++ specific optimizations in the matrix representation algorithm.

5 Conclusions

We have created CVXcanon, an open source library to unify the matrix representation step of canonicalization. The performance benefits of CVXcanon have been demonstrated empirically in CVXPY. More generally, CVXcanon is the first step towards creating a unified backend for all of the CVX.* systems in C++. Towards this broad goal, there are a number of directions for future work.

- *Convex.JL and CVX Support* A natural next step is extending Convex.JL and CVX to perform canonicalization using CVXcanon. In both languages, the main challenge is modifying the Julia and Matlab canonicalization code to explicitly build linear expression trees and use the same linear operation data structure as CVXPY and CVXcanon.
- *Solver Support* After canonicalization, CVXcanon returns the problem data to CVXPY via a wrapper and CVXPY immediately calls a backend solver via another wrapper. It would be more efficient and robust to avoid repeatedly passing data through wrappers and instead call the solver directly after constructing the matrix in C++.

6 Acknowledgements

We thank Steven Diamond for the initial project idea and guidance throughout the project.

References

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[GB08] Michael Grant and Stephen Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008.

[GB14] Michael Grant and Stephen Boyd. CVX: Matlab Software for Disciplined Convex Programming, version 2.1. `http://cvxr.com/cvx`, March 2014.

[Gra04] M. Grant. *Disciplined Convex Programming*. PhD thesis, Stanford University, 2004.

[MUB] D. Zeng J Hong S. Diamond M. Udell, K. Mohan and S. Boyd. Convex Optimization in Julia. *Proceedings High Performance Technical Computing in Dynamic Languages (HPTCDL)*.

[SDB14] E. Chu S. Diamond and S. Boyd. CVXPY: A Python-Embedded Modeling Language for Convex Optimization, version 0.2. `http://cvxpy.org/`, May 2014.

A Build Matrix Algorithm

The function `build_matrix` creates our problem matrix by traversing the linear atom tree.

```
define build_matrix:
given a list linear atoms $L$ and a map from variables to column numbers, $M$
Let $m$ be the total number of variables and $n$ be the total length of our constraints
Initialize $A \in \mathbf{R}^{m \times n}$ and $b \in \mathbf{R}^n$
Initialize $v$, our vertical offset, to zero
for each $\ell$ in $L$:
    Let $C =$ get_coefficients($\ell$), a list of (matrix, id) tuples
    for each block $B$, variable id id in C
        if $id$ is CONSTANT, squish $B$, and add to $b$ at offset $v$
        else Add $B$ to $A$ at horizontal offset $M[id]$ and vertical offset $v$
return $A$ and $b$
```

The function `build_matrix` calls the recursive function `get_coefficient`, which populates a populates a list of matrices along with their associated variable ID, which can be used to determine their horizontal offset within the problem matrix. Their vertical offset is given determined by which constraint they encode.

```
define get_coefficient:
given a linear atom $\ell$
Let $L :=$ get_root_coefficients($\ell$)
If $\ell$ has variable type or constant type return L
initialize an empty list of (matrix, variable id) tuples, coeffs
for each child $c$ of $\ell$
    Let $G :=$ get_coefficient(c)
    for each matrix $R$, id, in $G$, add ($LR$, $id$ ) to $C$
return C
```

`get_root_coefficient` has special cases for each of our linear atom types. For more details on this cases, refer to `LinOpOperations.cpp` in the CVXcanon repository.

B CVXPY Problems

Note the CVXcanon experiments were run by adding `settings.USE_CVXCANON = True` to the source code at the start of each function below.

Summation.

```
n = 10000
x = Variable()
e = 0
for i in range(n):
    e = e + x
p = Problem(Minimize(norm(e-1,2)), [x>=0])
p.get_problem_data(ECOS)
```

Indexing.

```
n = 10000
x = Variable(n)
e = 0
for i in range(n):
    e += x[i];
p = Problem(Minimize(norm(e-1,2)), [x>=0])
p.get_problem_data(ECOS)
```

Transpose.

```
n = 500
A = numpy.random.randn(n,n)
X = Variable(n,n)
p = Problem(Minimize(norm(X.T-A,'fro')), [X[1,1] == 1])
p.get_problem_data(ECOS)
```

Matrix Constraint.

```
n = 500
A = numpy.random.randn(n,n)
B = numpy.random.randn(n,n)x
X = Variable(n,n)
p = Problem(Minimize(norm(X-A,'fro')), [X == B])
p.get_problem_data(ECOS)
```

Matrix Product.

```
n = 50
A = numpy.random.randn(n,n)
X = Variable(n,n)
p = Problem(Minimize(norm(X,'fro')), [A.T*X*A >= 1])
p.get_problem_data(ECOS)
```

SVM with Indexing.

```
def gen_data(n):
    pos = numpy.random.multivariate_normal([1.0,2.0],numpy.eye(2),size=n)
    neg = numpy.random.multivariate_normal([-1.0,1.0],numpy.eye(2),size=n)
    return pos, neg

N = 2
C = 10
```

```
pos, neg = gen_data(500)

w = Variable(N)
b = Variable()
xi_pos = Variable(pos.shape[0])
xi_neg = Variable(neg.shape[0])
cost = sum_squares(w) + C*sum_entries(xi_pos) + C*sum_entries(xi_neg)
constrs = []
for j in range(pos.shape[0]):
    constrs += [w.T*pos[j,:] - b >= 1 - xi_pos[j]]

for j in range(neg.shape[0]):
    constrs += [-(w.T*neg[j,:] - b) >= 1 - xi_neg[j]]
p = Problem(Minimize(cost), constrs)
p.get_problem_data(ECOS)
```