CVX.jl: A Convex Modeling Environment in Julia

Jenny Hong       Karanveer Mohan       David Zeng

June 4, 2014

Abstract

CVX.jl is a modeling environment for disciplined convex programming (DCP) in Julia, an open-source scientific computing language. CVX.jl parses human-readable, DCP-compliant expressions into standard forms for backend solvers. Currently, CVX.jl is capable of solving linear programs (LPs) and second-order cone programs (SOCPs), using Embedded Conic Solver (ECOS) for its backend.

1 Introduction

1.1 Motivation

Julia is an open-source scientific computing language that boasts the performance of C, the dynamism of Ruby, and the mathematical prowess of Matlab. Though nascent, Julia aims to be as easy to use as Python and Matlab while outclassing them in speed. A native convex solver would both expand development in Julia and test its computational power.

1.2 Background

CVX.jl was started by Madeleine Udell, who had initially written a minimalistic wrapper around CVXPY [DCB14]. We have reworked CVX.jl to instead directly use a backend C solver. Our current codebase is available at https://github.com/cvxgrp/CVX.jl.

Our main technical challenge is more in software engineering than in mathematical ingenuity. This is an open-source project, so we have especially prioritized documentation and code quality. We began by implementing a framework for LPs, but our structure is robust enough that we have now implemented SOCPs and will move on to SDPs and mixed-integer programs.

2 Basic Usage

We first introduce the basic functionality of CVX.jl with example code. More thorough documentation, including installation instructions, is available in the README of our GitHub
In addition, several usage examples are available in the `test` and `examples` directories.

## 2.1 Example Code

- Declaring variables:
  
  To declare variables \( x \in \mathbb{R}, y \in \mathbb{R}^4, \) and \( Z \in \mathbb{R}^{4 \times 6}, \) we use the following syntax.

  ```
  x = Variable()
  y = Variable(4)
  Z = Variable(4, 6)
  ```

- Forming expressions:
  
  We can form expressions from CVX.jl variables and expressions as well as regular Julia constants.

  ```
  expr1 = sum_squares(y + 2)
  expr2 = Z[1:4] + x
  ```

- Forming constraints:

  ```
  constr1 = 10 * expr1 <= sum(x) + sum(reshape(Z, 3, 8))
  constr2 = expr2 >= 3
  ```

- Creating problems:

  Constraints can be added at construction or appended later.

  ```
  problem = minimize(norm_2(y), [constr1, constr2])
  problem.constraints += (y >= 0)
  ```

- Solving problems:

  ```
  solve!(problem)
  ```

  In Julia, the `!` after a function name is a convention that indicates that the arguments will be modified. In this case, `solve!` populates relevant fields (e.g., `problem.optval`) and values of variables (e.g., `x.value`).

- Extracting optimal values (of the problem, primal variables, user-defined expressions, and dual variables):

  ```
  println(problem.optval)
  println(x.value)
  println(expr1.evaluate())
  println(problem.constraints[1].dual_value)
  ```
2.2 Functionality
CVX.jl currently supports linear programs and second-order cone programs with scalar, vector, or matrix variables. Supported operations, also known as atoms, include

- Arithmetic atoms: +, -, *, /
- Slicing and shaping atoms: getindex, hcat, vertcat, reshape, diag, vec, transpose
- Positive orthant atoms: sum, abs, max, min, pos, neg, norm_1, norm_inf
- SOC atoms: norm_2, norm_fro, square, sqrt, geo_mean, quad_over_lin, inv_pos, sum_squares, square_pos, qol_elementwise.

For more details, please refer to the cvx user guide [GB13].

3 Implementation Details
We ultimately store user-generated information about the problem in a Problem instance, which contains an objective and a list of constraints. The objective has a symbol (i.e. minimize, maximize, or satisfy), and for maximization and minimization problems, the objective is also associated with an expression.

3.1 Expression Types
All expressions in CVX.jl are one of the three subtypes of AbstractCvxExpr.

- **Constant**: Wrappers around constant-valued scalars, vectors, or matrices.
- **Variable**: User-declared variables of the optimization problem.
- **CvxExpr**: General expressions that depend on other expressions, variables, or constants.

Each expression type keeps track of its convexity, sign, size, and canonical form (see next section for more information about canonicalization). Constants and Variables can be declared directly. CvxExprs are formed using atoms, which are functions that take in AbstractCvxExprs as arguments. Refer to section 2.2 for a list of atoms.

3.2 Syntax Tree
The Problem type and its members recursively form a syntax tree. To see the mechanics of the syntax tree, consider the simple optimization problem

\[
\begin{align*}
\text{minimize} & \quad \|x\|_\infty \\
\text{subject to} & \quad x_1 + x_2 = 5 \\
& \quad x_3 \leq x_2,
\end{align*}
\]
where \( x \in \mathbb{R}^3 \) is the optimization variable. CVX.jl parses this problem into the following expression tree:

![Expression Tree]

In this example, the minimization objective is the \texttt{CvxExpr} returned by the \texttt{norm.inf} atom, whose argument is a single \texttt{Variable}.

Each user-declared constraint is of type \texttt{CvxConstr}, which holds an operator (i.e. \( \leq, =, \) or \( \geq \)), a left-hand side \texttt{AbstractCvxExpr}, and a right-hand side \texttt{AbstractCvxExpr}. In this example, the first \texttt{CvxConstr} we encounter has operator \( = \). Its LHS is a \texttt{CvxExpr} and its RHS is a \texttt{Constant}.

### 3.3 Canonicalization

The constraints and objective of the problem must be formulated into a canonical form for the backend solver. The ECOS backend expects the following canonical form:

\[
\begin{align*}
\min & \quad c^T x \\
\text{subject to} & \quad Ax = b \\
& \quad Gx \preceq h
\end{align*}
\]

where \( K = \mathbb{R}_+^l \times Q_{n_1} \times \ldots \times Q_{n_N} \) and \( Q_p = \{(t, s) \in \mathbb{R} \times \mathbb{R}^{p-1} \mid \|s\|_2 \leq t\} \) is the second-order cone of dimension \( p \).

#### 3.3.1 Canonicalizing Constraints

A constraint such as

\[
\text{norm}_2(A \ast z + b) \leq 3
\]
with variable \( z \in \mathbb{R}^n \) and data \( A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m \) will be parsed into expressions with the following relationships:

\[
\begin{align*}
  t_0 &\leq 3 \tag{1} \\
  t_0 &= \|t_1\|_2 \tag{2} \\
  t_1 &= t_2 + b \tag{3} \\
  t_2 &= Az \tag{4}
\end{align*}
\]

where unique identifiers \( t_0, t_1, t_2 \) have been introduced to represent each expression \[CPDB13\]. These relationships between the expressions implicitly introduce additional constraints (2), (3), and (4) to the problem. These constraints can then be collected together to build the affine equality constraint \( Ax = b \) and the conic inequality constraint \( Gx \preceq_K h \) in the following manner:

\[
\begin{bmatrix}
  0 & -I_m & I_m & 0 \\
  0 & 0 & -I_m & A \\
\end{bmatrix}
\begin{bmatrix}
  t_0 \\
  t_1 \\
  t_2 \\
  z
\end{bmatrix} =
\begin{bmatrix}
  b \\
  0
\end{bmatrix}
\tag{5}
\]

\[
\begin{bmatrix}
  1 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 \\
  0 & -I_m & 0 & 0
\end{bmatrix}
\begin{bmatrix}
  t_0 \\
  t_1 \\
  t_2 \\
  z
\end{bmatrix} \succeq \mathbb{R}_{m \times q_{m+1}}
\tag{6}
\]

These implicit constraints that arise during canonicalization (e.g., (5) and (6)) are stored in a CanonicalConstr instance in every CvxExpr and CvxConstr. Each CvxExpr and CvxConstr stores both its own implicit canonicalization constraints and those of all its descendants. Each CvxExpr and CvxConstr has a method called canon_form(), which returns its canonical form.

### 3.3.2 Canonicalizing Matrix Variables

The canonical form supports only one vector variable \( x \), so matrix variables need to be vectorized. For \( X \in \mathbb{R}^{m \times n} \), the identities \[Fac05\]

\[
\begin{align*}
  \text{vec}(AX) &= (I_n \otimes A)\text{vec}(X) \\
  \text{vec}(XB) &= (B^T \otimes I_m)\text{vec}(X)
\end{align*}
\]

allow us to perform multiplication on the vectorized form of \( X \). Here, \( \otimes \) denotes the Kronecker product.

### 3.4 Backend Solvers

Julia provides functionality to directly call C code compiled as shared libraries. However, any C code that returns a C struct or a pointer to a struct needs a wrapper around the
function calls to convert the C struct to an equivalent Julia object. Similarly, Julia objects
must be converted into C structs before they can be passed in as arguments to C functions.

CVX.jl currently uses ECOS, a solver for second-order cone programs. To interface with ECOS, we have forked ECOS.jl [San14] and revised it to be compatible with CVX.jl while maintaining the necessary modularity. Our fork is available at [https://github.com/karanveerm/ECOS.jl](https://github.com/karanveerm/ECOS.jl).

We have also written and packaged a Julia wrapper around Splitting Cone Solver (SCS), which can be found at [https://github.com/karanveerm/SCS.jl](https://github.com/karanveerm/SCS.jl), in preparation for extending the functionality of CVX.jl in the future.

## 4 Further Work

We will add more performance tests, particularly to compare with CVX and CVXPY benchmarks, and continue to expand the existing documentation.

### 4.1 Backend Solvers

By switching CVX.jl to use SCS instead of ECOS as its backend solver, we will be able to support exponential and semidefinite cone constraints, and atoms such as \( \log \text{sum.exp} \), \( \log \text{det} \), \( \text{norm.nuclear} \), \( \text{kl.divergence} \), \( \lambda_{\text{max}} \), and \( \lambda_{\text{min}} \).

The Julia developer community is currently building MathProgBase.jl [LHDB14], a library of unified interfaces for Julia modeling frontends to communicate with backend solvers. MathProgBase aims to define these interfaces for LPs, cone programs, mixed-integer programs, and others. Ensuring CVX.jl is compatible with these interfaces will allow us to easily use a variety of backend solvers such as Gurobi, GLPK, etc.

### 4.2 Reductions

We plan to perform reductions to prune our expression tree. For example, for the expression \( 4 \times x + 3 \times x \), we currently form three additional identifiers, one for each of the multiplications and a third for the addition. Instead, we could recognize that an equivalent expression is \( 7 \times x \) and use only one additional identifier to represent the expression.

## Acknowledgments

We thank Madeleine Udell for the project idea, for her work on starting CVX.jl, and for her continued support and direction. We also thank Stephen Boyd for his mentorship. Finally, we thank Steven Diamond for his experienced guidance as the author of CVXPY, which has been a very useful model for us.
References


