# Convex Optimization in Julia

Madeleine Udell
udell@stanford.edu

Karanveer Mohan
kvmohan@stanford.edu

David Zeng
dzeng0@stanford.edu

Jenny Hong
jyunhong@stanford.edu

Steven Diamond
stevend2@stanford.edu

Stephen Boyd
boyd@stanford.edu

## ABSTRACT

This paper describes `Convex`[1], a convex optimization modeling framework in Julia. `Convex` translates problems from a user-friendly functional language into an abstract syntax tree describing the problem. This concise representation of the global structure of the problem allows `Convex` to infer whether the problem complies with the rules of disciplined convex programming (DCP), and to pass the problem to a suitable solver. These operations are carried out in Julia using multiple dispatch, which dramatically reduces the time required to verify DCP compliance and to parse a problem into conic form. `Convex` then automatically chooses an appropriate backend solver to solve the conic form problem.

## Categories and Subject Descriptors

G.1.6 [**Numerical Analysis**]: Optimization—*Convex Programming*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

## General Terms

Languages

## Keywords

Convex programming, automatic verification, symbolic computation, multiple dispatch

## 1. INTRODUCTION

The purpose of an optimization modeling language is to translate an optimization problem from a user-friendly language into a solver-friendly language. In this paper, we present an approach to bridging the gap. We show that mathematical optimization problems can be parsed from a simple human-readable form that uses the trope of function composition, into an abstract syntax tree (AST) representing the problem.

[1]available online at `http://github.com/cvxgrp/Convex.jl`

Global information about the expression is retained in the AST. Thus, expressions can be checked for convexity by applying the rules of *disciplined convex programming*, pioneered by Michael Grant and Stephen Boyd in [20, 19]. The AST can be used to convert the problem into a conic form optimization problem, allowing a solver access to a complete and computationally concise global description of the problem [37].

### *Julia.*

The Julia language [5] is a high-level, high-performance dynamic programming language for technical computing. With a syntax familiar to users of other technical computing languages such as Matlab, it takes advantage of LLVM-based just-in-time (JIT) compilation [26] to approach and often match the performance of C [4]. In `Convex`, we make particular use of *multiple dispatch* in Julia, an object-oriented paradigm in which methods are defined on combinations of data types (classes), instead of being encapsulated inside classes [3].

The `Convex` project supports the assertion that " technical computing is [the] killer application [of multiple dispatch]" [3]. We show in this paper that Julia's multiple dispatch allows the authors of a technical computing package to write extremely performant code using a high level of abstraction. Indeed, the abstraction and generality of the code has pushed the authors of this paper toward more abstract and general formulations of the mathematics implemented in the package than have been previously published, while producing code whose performance rivals and often surpasses codes with similar functionality in other languages.

Moreover, multiple dispatch enforces a separation of the fundamental objects in `Convex`— functions — from the methods for operating on them. This separation has other benefits: in `Convex`, it is easy to implement new methods for operating on the AST of an optimization problem, opening the door to new structural paradigms for parsing and solving optimization problems.

### *Mathematical optimization.*

A traditional and familiar form for specifying a mathematical optimization problem (MOP) is to write

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i=1,\ldots,m \\ & h_i(x) = 0, \quad i=1,\ldots,p, \end{array}$$

with variable $x \in \mathbf{R}^n$. Here, a problem instance is specified as list of *functions* $f_0,\ldots,f_m,h_1,\ldots,h_p : \mathbf{R}^n \to \mathbf{R}$. The function $f_0$ is called the objective; $f_1 \ldots, f_m$ are called the inequality constraints; and $h_1, \ldots, h_p$ are called the equality constraints. The field of mathematical optimization is concerned with finding methods for solving MOP.

Some special structural forms of MOP are well known. For example, if all the functions $f_0, \ldots, f_m, h_1, \ldots, h_p$ are affine, the MOP is called a linear program (LP). When each of the functions $f_0, \ldots, f_m$ is convex, and $h_1, \ldots, h_p$ are all affine, it is called a convex program (CP). This paper focusses on solving CPs, which can be solved much more quickly than general MOPs [23].

Perhaps the most consistent theme of the convex optimization literature concerns the importance of using the *structural form* of the CP (that is, the properties of the functions $f_0, \ldots, f_m, h_1, \ldots, h_p$) in order to devise faster solution methods. The advantage of this structural approach is that it allows for a division of labor between *users* of optimization who specialize in modeling real-world problems using optimization, and *designers* of optimization algorithms. Designers of optimization algorithms write solvers specialized to a particular structural form of MOP. Users can take advantage of innovations in optimization algorithms and theory so long as they can *identify* the structural form of their problem, and apply the appropriate solver.

A number of general purpose solvers have been developed, including AMPL [15], GAMS [8, 9], ACADO [22, 21, 2], Ipopt [6], and NLopt [23]), along with associated modeling languages that make it easy to give problems to these solvers, including AMPL [15], GAMS [8, 9], and JuMP [30]. However, these solvers can be significantly slower than ones designed for special problem classes [25, 34, 36].

### *Cone programs.*

`Convex` targets a special class of problems called *cone programs*, which includes LPs, QPs, QCQPs, SOCPs, SDPs, and exponential cone programs as special cases (see §4 for more details). Cone programs occur frequently in a variety of fields [28, 44, 45], can be solved quickly, both in theory and in practice [38].

A number of modeling systems have been developed to automatically perform translations of a problem into standard conic form. The first parser-solver, SDPSOL, was written using Bison and Flex in 1996, and was able to automatically parse and solve SDPs problems with a few hundred variables [46]. YALMIP [29] was its first modern successor, followed shortly thereafter by CVX [20], both of which embedded a convex optimization modeling language into MATLAB, a proprietary language. Diamond et al followed with CVXPY [13], a convex optimization modeling language in python which uses an object-oriented approach. The present paper concerns `Convex`, which borrows many ideas from CVXPY, but takes advantage of language features in Julia (notably, multiple dispatch) to make the modeling layer simpler and faster.

A few more specialized conic parsing/modeling tools have also appeared targeting small QPs and SOCPs with very tight performance requirements, including CVXGEN [32] and QCML [10].

### *Organization.*

The organization of this paper proceeds as follows. In §2, we discuss how `Convex` represents optimization problems, and show how this representation suits the requirements of both humans and solvers. In §3, we show how to use the AST to verify convexity of the problem using the rules of disciplined convex programming. In §4, we show how to use the AST to transform the problem into conic form, and pass the problem to a conic solver. Finally, in §5 we compare the performance of `Convex` to other codes for disciplined convex programming.

## 2. FUNCTIONAL REPRESENTATION

In this section, we describe the functional representation used by `Convex`. The basic building block of `Convex` is called an *expression*. We explain some types of expressions and how they can be combined to create more complex ones.

### *Variables.*

The simplest kind of expression in `Convex` is a variable. Variables in `Convex` are declared using the `Variable` keyword, along with the dimensions of the variable.

```
# Scalar variable
x = Variable()

# 4x1 vector variable
y = Variable(4)

# 4x2 matrix variable
z = Variable(4, 2)
```

Variables may also be declared as having special properties, such as being positive, negative, or positive semidefinite matrices. These properties are treated as constraints in any problem constructed using the variables.

```
# Positive scalar variable
x = Variable(Positive())

# Negative 4x1 vector variable
y = Variable(4, Negative())

# Symmetric positive semidefinite
# 4x4 matrix variable
z = Semidefinite(4)
```

Every variable that is created is assigned a global unique identifier.

### *Constants.*

Numbers, vectors, and matrices present in the Julia environment are wrapped automatically into a `Constant` expression when used in `Convex`. The `Constant` expression stores a pointer to the underlying value, so the expression will reflect changes made to the value after the expression is formed. In other words, `Convex` expressions are parametrized by the constants they hold.

### *Atoms.*

To form more complex expressions, we use elements called *atoms* such as `+`, `*`, `abs`, and `norm`, which represent simple functions. Function composition is implemented in `Convex` by applying atoms to expressions. A list of some of the atoms available in `Convex` is given in Table 3.

Atoms are applied to expressions using operator overloading. Hence, `2+2` calls Julia's built-in addition operator, while

2+x calls the Convex addition method and returns a Convex expression. Many of the useful language features in julia, such as arithmetic, array indexing, and matrix transpose are overloaded in Convex so they may be used with variables just as they are used with constants.

For example, all of the following form valid expressions.

```
# indexing, multiplication, addition
e1 = y[1] + 2*x

# expressions can be affine, convex, or concave
e3 = sqrt(x)

# more atoms
e2 = 4 * pos(x) + max(abs(y)) + norm(z[:,1],2)
```

Some atoms are defined by composing other atoms; we call these *composite* atoms. For example, typing norm(x, 1) calls sum(abs(x)). Each basic (non-composite) atom has a number of methods defined on it that allow Convex to identify the convexity of expressions formed by applying the atom, and to transform the atom into standard conic form.

### *Expressions.*

Formally, expressions are defined recursively to be a variable, a constant, or the result of applying a atom to one or more expressions. In Convex, each expression is represented as an AST with a variable or constant at each leaf node, and a atom (function) at each internal node. We refer to the arguments to the atom at a certain node as the *children* of that node. The top-level atom is called the *head* of the AST.

Considered as a tree without leaves, the expression may be seen as representing a function formed by composing the functions which the atoms represent; with the leaves, it may be interpreted as a function with a closure, a curried function, or simply a function of named arguments. The AST is a directed acyclic graph (DAG) [16]. The idea of using a DAG to study global properties of optimization problems dates back to work by Kantorovich in the 1950s [24]. Note that the AST does not depend on the solver format we target, or on the convexity of the expressions.

Expressions have a number of properties, including value, sign, size, curvature, and range, which may be left undefined when the expression is first formed. Expressions may be evaluated when every leaf node in their AST has already been assigned a value. For example,

```
x = Variable()
e = max(x,0)
x.value = -4
evaluate!(e)
```

evaluates to 0.

One novel aspect of Convex is that expressions are identified via a unique id computed as a hash on the top-level atom name and the unique ids of its children in the AST. Thus, two different instances of the same AST will automatically be assigned the same unique id. This reduces the complexity of the problem that is ultimately passed to the solver [16].

### *Constraints.*

*Constraints* in Convex are declared using the standard comparison operators <=, >=, and ==. They specify relations that must hold between two expressions. Convex does not distinguish between strict and non-strict inequality constraints.

```
# affine equality constraint
A = randn(3,4); b = randn(3)
c1 = A*y == b

# convex inequality constraint
c2 = norm(y,2) <= 2
```

### *Problems.*

A *problem* in Convex is composed of a *sense* (minimize, maximize, or satisfy), an objective (an expression to which the sense verb is to be applied), and a list of zero or more constraints which must be satisfied at the solution.

For example, the following code forms the nonnegative least squares problem

$$\begin{array}{ll} \text{minimize} & \|x\|_\infty \\ \text{subject to} & x_1 + x_2 = 5 \\ & x_3 \le x_2, \end{array}$$

where $x \in \mathbf{R}^3$ is the optimization variable.

```
x = Variable(3)
constraints = [x[1]+x[2] == 5, x[3] <= x[2]]
p = minimize(norm_inf(x), constraints)
```

When the user types this code, the input problem description is parsed internally into a Problem object with two attributes: an objective and a list of constraints. Each expression appearing in the objective is represented as an AST with one atom at each internal node.

```
p.objective   = (:norm_inf, x)
p.constraints = [(:==, 5, (:+, (:getindex, x, 1),
                               (:getindex, x, 2))),
            (:<=, getindex(x, 3), getindex(x, 2)]
```

Figure 1 shows the structure of this problem.

### *Solving problems.*

In the next sections, we will see how the information stored in the AST can be used to verify that the problem is a disciplined convex program (defined below in §3), and to convert the problem into a standard conic form program (defined below in §4). The solve! method in Convex checks that the problem is a disciplined convex program, converts it into conic form, and passes the problem to a solver.

```
solve!(p)
```

The solve! method mutates its argument. After calling it, the optimal value of p can be queried with p.optval, and every variable x in p is annotated with a value, accesible via x.value. If the solver computes dual variables, these are populated in the constraints; for example, the dual variable corresponding to the first constraint can be accessed as p.constraints[1].dual_value.

## 3. DISCIPLINED CONVEX PROGRAMMING

Checking if a function is convex is a difficult problem in general. Many approaches to verifying convexity have been

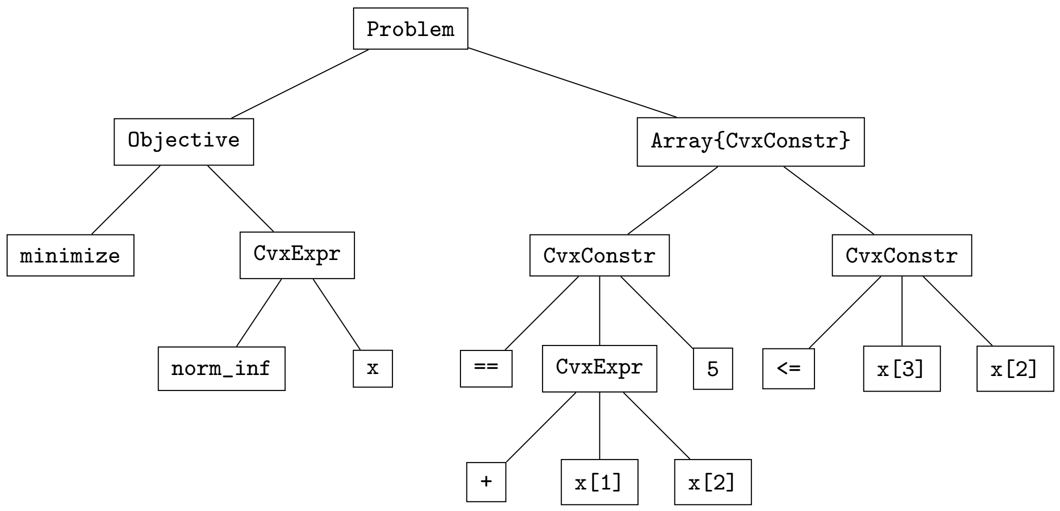

**Figure 1: Expression tree representation of a problem**

proposed [41, 12], and implemented in modeling systems ranging from AMPL to the solver in Microsoft Excel [16, 35, 17]. Here, we have chosen to use the framework of *disciplined convex programming* (DCP) to verify problem convexity, which has a number of advantages. First, the simplicity of the DCP rule set makes the determination of convexity transparent to the user; it forms an easily "teachable" system [7]. Second, a problem that is DCP compliant (or more simply, DCP) can be converted to an equivalent conic form problem. Hence `Convex` is able to take advantage of extremely fast and reliable solvers for any problem whose convexity can be verified using these rules.

### *Disciplined convex expressions.*

The rules of disciplined convex programming define disciplined convex expressions inductively. The simplest expressions are single variables, which are defined to be affine, and constants, which are defined (unsurprisingly) to be constant. Now suppose that the convexity of the expressions $e_1, \dots, e_n$ are known. Then the expression $f(e_1, \dots, e_n)$ is convex if the function $f$ is convex on the range of $(e_1, \dots, e_n)$, and for each $i = 1, \dots, n$,

- $f$ is nondecreasing in its $i$th argument over the range of $(e_1, \dots, e_n)$, and $e_i$ is convex;
- $f$ is nonincreasing in its $i$th argument over the range of $(e_1, \dots, e_n)$, and $e_i$ is concave; or
- $e_i$ is affine.

The expression $f(e_1, \dots, e_n)$ is concave if $(-f)(e_1, \dots, e_n)$ is convex, and it is affine iff it is both convex and concave.

We say that an expression is DCP compliant if its curvature (affine, convex, or concave) can be inferred from these rules. Hence in order to decide if any AST is DCP, we need only be able to calculate the curvature of $f$ over a range, and the monotonicity of $f$ in each argument over a range, for any atom $f$.

The set of DCP compliant functions clearly depends on the allowed atoms. If a function is known to be convex, but its convexity cannot be derived from the DCP rules, it can be added to the list of atoms and used in turn to build more complex functions. For example, $\log(\sum_{i=1}^n \exp x_i)$ is convex, but its convexity cannot be derived from the DCP rules using only the properties of log and exp [19]. In `Convex`, it is easy to add new atoms to the library of known atoms, allowing sophisticated users to expand the set of functions that `Convex` can recognize as convex.

Determining that an expression is DCP compliant also depends on what is known about the monotonicity of the atom as a function of its arguments. For example, the quadratic-over-linear function $f(x, y) = x^T x/y$ is not convex on $\mathbf{R}^2$, but it is convex on $\mathbf{R} \times (0, \infty)$. This observation is known as a *signed* convexity rule [13]; our formulation of the DCP rules above in terms of the *ranges* of the expressions $(e_1, \dots, e_n)$ generalizes this observation.

These simple rules provide an easy way to inductively determine the convexity of all expressions, so long as the atoms have a means of checking their monotonicity in each argument.

### *Verifying disciplined convex expressions using multiple dispatch.*

`Convex` is able to efficiently check that expressions are DCP using multiple dispatch. `Convex` defines for each atom two methods:

- `monotonicity` takes an atom and a list of arguments (expressions), and returns the *monotonicity* of the function the atom represents (nondecreasing, nonincreasing, or not monotonic) over the range of the arguments.
- `curvature` takes an atom and a list of arguments (expressions), and returns the curvature of the function the atom represents (convex, concave, affine, or neither) on the range of the arguments.

Arithmetic is defined directly on vexities and monotonicities so that addition of vexities and multiplication of vexities by monotonicities enforce the DCP rules. For example, our definitions make the following assertions true:

```
Affine() + Convex() == Convex()
```

```
Concave() + Convex() == NotDCP()
Nonincreasing() * Convex() == Concave()
NotMonotonic() * Affine() == Affine()
```

Vexity checking can then be implemented by applying the following short function:

```
function vexity(x::AbstractExpr)
  monotonicities = monotonicity(x)
  vexity = intrinsic_vexity(x)
  for i = 1:length(x.children)
    vexity += monotonicities[i]
                 * vexity(x.children[i])
  end
  return vexity
end
```

This approach to checking convexity has a number of advantages over using if-else statements to enforce the DCP rules. The implementation is aesthetically appealing: the code follows the mathematics. The convexity of arguments is checked only by the universal `vexity` function, rather than inside a method pertaining only to a single atom. This structure makes it easy to write new atoms, since one only needs to implement the `curvature` and `monotonicity` methods. It also reduces the time needed to verify convexity, since multiple dispatch is implemented as a lookup table rather than via a (slower) `if` statement.

### *Disciplined convex constraints.*

A constraint is called a *disciplined convex constraint* (or DCP constraint) if it has the form

- $e_1 \leq e_2$, where $e_1$ is a convex DCP expression and $e_2$ is a concave DCP expression;
- $e_1 \geq e_2$, where $e_1$ is a concave DCP expression and $e_2$ is a convex DCP expression; or
- $e_1 = e_2$, where $e_1$ and $e_2$ are both affine DCP expressions.

(For the purposes of this definition, remember that constant expressions are affine, and affine expressions are both convex and concave.)

### *Disciplined convex programs.*

A problem is called a *disciplined convex program* if

1. every constraint in the problem is DCP compliant; and
2. the (sense, objective) are
    - (minimize, convex DCP expression),
    - (maximize, concave DCP expression), or
    - (satisfy,).

(Note that the sense satisfy does not take an objective.) When an optimization problem is formed, `Convex` applies the DCP rules recursively to determine whether the problem is DCP.

## 4. CONIC FORM OPTIMIZATION

A conic form optimization problem is written as

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

with variable $x$. Here $\mathcal{K}$ is a *cone*: a set of points such that $x \in \mathcal{K}$ iff $rx \in \mathcal{K}$ for every $r \geq 0$. If in addition $\mathcal{K}$ is convex, it is called a convex cone.

Examples of convex cones include the following.

- *The zero cone.* $\mathcal{K}_0^1 = \{0\}$
- *The free cone.* $\mathcal{K}_{\text{free}}^n = \mathbf{R}^n$
- *The positive orthant.* $\mathcal{K}_+^n = \{x \in \mathbf{R}^n : x \geq 0\}$
- *The second order cone.* $\mathcal{K}_{\text{SOC}}^{n+1} = \{(x,t) \in \mathbf{R}^n : \|x\| \leq t\}$
- *The semidefinite cone.* $\mathcal{K}_{\text{SDP}}^{n^2} = \{X \in \mathbf{R}^{n\times n} : \lambda_{\min}(X) \geq 0, X = X^T\}$
- *The exponential cone.* $\mathcal{K}_{\text{exp}}^3 = \{(x,y,z) \in \mathbf{R}^3 : ye^{x/y} \leq z, y > 0\}$

In general, a cone $\mathcal{K}$ may also be given as a product of simpler cones,

$$\mathcal{K} = \mathcal{K}_1 \times \cdots \times \mathcal{K}_p.$$

A conic form optimization problem is specified by the problem data $A$, $b$, $c$, and the cone $K$. We assume the cone $K$ is given as a product of cones drawn from the six convex cones described above.

A number of fast solvers exist for conic form problems in a variety of languages. We list a number of these solvers in Table 4.

`Convex` is able to automatically rewrite a problem specified as a DCP in standard cone format, and to pick a cone solver automatically depending on the structure of the problem and the solvers available on the user's computer, using the `loadconicproblem!` interface in MathProgBase [30]. This allows `Convex` to, for example, choose a SOCP solver if no exponential or SDP cones are present in the problem, which can result in significantly faster solve time [1].

## 4.1 Example

Here, we describe how `Convex` transforms a DCP problem into conic form.

### *Conic form expression..*

Every function $f(x)$ can be thought of as the trivial optimization problem

$$\text{minimize} \quad f(x),$$

with no variable. For atoms that are cone-representable, we can also define the function as a (nontrival) optimization problem in standard conic form, by introducing new variables. For example, $|x|$ is transformed into the problem

$$\begin{array}{ll} \text{minimize} & t \\ \text{subject to} & t - x \in \mathcal{K}_+ \\ & t + x \in \mathcal{K}_+. \end{array}$$

In the standard conic form used by `Convex`, every expression is affine, and the constraints are all conic constraints.

In general, the conic form template of a function is a constructor for an optimization problem. It takes the arguments of the function, $x_1, \ldots, x_n$, and returns a sense, an objective and a (possibly empty) set of constraints $(s, o, \mathcal{C})$ in which every expression is affine in $x_1, \ldots, x_n$. The conic template of a function is also known as the *graph form* of the function [18]. For convex functions $f$, any feasible point $(x_1, \ldots, x_n, t)$ for the conic form problem is a point in the epigraph $\{(x,t) : f(x) \leq t\}$ of $f$; for concave functions, any feasible point of the conic form problem lies in the hypograph of $f$. Note that the objective function of the conic form problem may be vector or matrix valued. For example, the standard conic form for $f(x) = Ax$ is simply

$$\text{minimize} \quad Ax.$$

`Convex` uses the conic form templates of the atoms in an expression recursively in order to construct the standard conic form of a expression. Recall that every expression is an AST with a function (atom) as its top-level node. We refer to the arguments to the function at a certain node as the children of that node. The standard conic form of an expression $e$ is constructed recursively as follows:

- If $e$ is affine, return $(e, \emptyset)$.
- Otherwise:
  1. Compute the standard conic forms $(s_i, o_i, \mathcal{C}_i)$ for each child of the top-level node of $e$.
  2. Splice the objectives $(o_1, \ldots, o_n)$ into the conic form template for the top-level node, producing a conic form problem $(s, o, \mathcal{C})$.
  3. Return $(s, o, \mathcal{C} \cup (\cup_i \mathcal{C}_i))$.

### *Conic form problem..*

Now that we have defined the standard conic form for an expression, it is simple to define the standard conic form for an optimization problem. The standard conic form of an optimization problem, given as an objective and constraint set, is computed as follows:

1. Let $(s, o, \mathcal{C})$ be the standard conic form of the objective.
2. For each constraint in the constraint set,
   (a) Compute the standard conic forms $(s_l, o_l, \mathcal{C}_l)$ and $(s_r, o_r, \mathcal{C}_r)$ for the left and right hand sides of the constraint.
   (b) Add $\mathcal{C}_l$ and $\mathcal{C}_r$ to $\mathcal{C}$.
   (c) If the sense of the constraint is
      - $\leq$, add $o_r - o_l \in \mathcal{K}_+$ to $\mathcal{C}$;
      - $\geq$, add $o_l - o_r \in \mathcal{K}_+$ to $\mathcal{C}$;
      - $=$, add $o_r - o_l \in \mathcal{K}_0$ to $\mathcal{C}$.

   (d) Return $(s, o, \mathcal{C})$.

Notice that we have not used the sense of the conic forms of the arguments to an expression in constructing its conic form. This would be worrisome, were it not for the following theorem:

THEOREM 1 ([19]). *Let $p$ be a problem with variable $x$ and dual variable $\lambda$, and let $\Phi(p)$ be the conic form of the problem with variables $x$ and $t$ and dual variables $\lambda$ and $\mu$. Here we suppose $t$ and $\mu$ are the primal and dual variables, respectively, that have been introduced in the transformation to conic form. If $p$ is DCP, then any primal-dual solution $(x, t, \lambda, \mu)$ to $\Phi(p)$ provides a solution $(x, \lambda)$ to $p$.*

To build intuition for this theorem, note that in a DCP expression, a convex function $f$ will have objectives with the sense minimize spliced into argument slots in which $f$ is increasing, and objectives with the sense maximize spliced into argument slots in which $f$ is decreasing. At the solution, the coincidence of these senses, monotonicity, and curvature will force the variables participating in the conic form of each atom to lie *on* the graph of the atom, rather than simply in the *epigraph*. Using this reasoning, the theorem can be proved by induction on the depth of the AST.

## 4.2 Solvers

Julia has a rich ecosystem of optimization routines. The `JuliaOpt` project collects mathematical optimization solvers, and interfaces to solvers written in other languages, in a single GitHub repository, while `MathProgBase` enforces consistent interfaces to these solvers. Through integration with `MathProgBase`, `Convex` is able to use all of the solvers that accept problems in conic form, which includes all the linear programming solvers in `JuliaOpt` (including GLPK [31] and the commercial solvers CPLEX [11], Gurobi [11], and Mosek [33]), as well as the open source interior-point SOCP solver ECOS [14], and the open source first-order primal-dual conic solver SCS [39]. A list of solvers which can be used with `Convex` is presented in Table 4.

## 5. SPEED

Here we present a comparison of `Convex` with CVXPY[2] [13] and CVX[3] [20] on a number of representative problems. The problems are chosen to be easy to *solve* (indeed, they are all easily solved by inspection) but difficult to *parse*. We concentrate on problems involving a large number of affine expressions; other atoms are treated more similarly in the different frameworks. The code for the tests in the three languages may be found in Appendix A.

We compare both the time required to convert the problem to conic form, and the time required to solve the resulting problem, since in general the conic form problems produced in different modeling frameworks may be different. We use all three modeling languages with the solver ECOS [14], and call the solver with the same parameters (`abstol=1e-7`, `reltol=1e-7`, `feastol=1e-7`, `maxit=100`) so that solve times are comparable between the modeling frameworks.

Parse times are presented in Table 1 and solve times in Table 2. Parse times are computed by subtracting solve time from the total time to form and solve the problem.

We present two different timings for `Convex`. Julia uses just in time (JIT) compilation; the code is compiled at the first evaluation, and the faster compiled version is used on subsequent calls, so the first time a function is called is in

[2]available at `https://pypi.python.org/pypi/cvxpy` as of October 5, 2014

[3]CVX v3.0 beta, available at `www.cvxr.com` as of October 5, 2014

**Table 1: Speed comparisons: parse time (s)**

| Test | CVX | CVXPY | `Convex` | `Convex` compiled |
|---|---|---|---|---|
| sum | 2.29 | 4.45 | 2.85 | 0.29 |
| index | 3.62 | 9.69 | 11.7 | 9.15 |
| transpose | 1.24 | 0.55 | 0.42 | 0.07 |
| matrix constraint | 1.38 | 0.54 | 2.76 | 0.33 |

**Table 2: Speed comparisons: solve time (s)**

| Test | CVX | CVXPY | `Convex` | `Convex` compiled |
|---|---|---|---|---|
| sum | 0.01 | 5e-4 | 2e-4 | 8e-5 |
| index | 0.01 | 0.08 | 0.23 | 0.19 |
| transpose | 0.66 | 1.54 | 0.45 | 0.43 |
| matrix constraint | 0.66 | 2.20 | 3.77 | 5.14 |

general slower than the second time. The 3rd column shows the runtime of the first (uncompiled) evaluation, and the 4th column the second (compiled) time. While the perfomance of `Convex` is comparable to CVX and CVXPY at first evaluation, it substantially outperforms the other packages upon the second evaluation in 3 out of the 4 tests. For applications in which convex optimization routines are called in an *inner* loop of a larger optimization algorithm (for example, in sequential convex programming), users will see the fast performance on every iteration of the loop after the first.

## 6. DISCUSSION

### *Flexibility.*

One of the principle advantages of the multiple dispatch paradigm is the ease with which new methods can be added. For example, one can simply add a new overloaded method to do any of the following tasks:

- add new composite atoms
- add new noncomposite atoms
- modify conic form
- add new backends

### *Future work.*

There are many other problem structures that may also be of interest. In general, any problem structure that has been identified and for which there are specialized, fast solvers may be a good candidate for a target problem form. Such problem structures include integer linear programs (which have received much attention from OR researchers for decades); biconvex problems (which may be heuristically solved by alternating minimization); sums of prox-capable functions (which can be efficiently solved by splitting methods like ADMM, even when the problem includes spectral functions of large matrices); difference-of-convex programs (for which sequential convex programming often produces successful solutions in practice) [27, 43]; and sigmoidal programming problems (which often admit fast solutions via a specialized branch and bound method); to name only a few. The number of structural forms that might be useful far exceeds the number that a potential consumer of optimization might care to remember.

The framework developed in `Convex` makes it easy to write new transformations on the AST to detect and transform these problem types into a standard format. Automatic detection of some of the problem types mentioned above is the subject of ongoing research.

Significant work also remains to allow `Convex` to handle extremely large scale problems. For example, `Convex` currently performs all computations serially; developments in Julia's parallelism ecosystem, and particularly its nascent threading capabilities, will enable `Convex` to leverage parallelism.

## 7. CONCLUSIONS

This paper shows that a convex optimization modeling library may be efficiently and simply implemented using multiple dispatch. The software package `Convex` uses language features of Julia (notably, multiple dispatch and JIT compilation) to parse DCP problems into conic form as fast or faster than other (significantly more mature) codes for modeling convex optimization.

## 8. REFERENCES

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

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

[3] J. Bezanson, J. Chen, S. Karpinski, V. Shah, and A. Edelman. Array operators using multiple dispatch: a design methodology for array implementations in dynamic languages. *arXiv preprint arXiv:1407.3845*, 2014.

[4] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. The julia language.

[5] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[6] L. T. Biegler and V. M. Zavala. Large-scale nonlinear programming using ipopt: An integrating framework

**Table 3: Atoms in `Convex`**

| | Function | Atom | Monotonicity | Curvature |
|---|---|---|---|---|
| Slicing and shaping atoms | | | | |
| | $f(x,i) = x_i$ | `getindex(x,i)` | nondecreasing | Affine |
| | $f(x,y) = [x\ y]$ | `hcat(x,y)` | nondecreasing | Affine |
| | $f(x,y) = [x, y]$ | `vertcat(x,y)` | nondecreasing | Affine |
| | $f(X) = (X_{11}, \ldots, X_{nn})$ | `diag(X)` | nondecreasing | Affine |
| | $f(X) = X^T$ | `transpose(X)` | nondecreasing | Affine |
| Positive orthant atoms | | | | |
| | $f(x) = \sum_i x_i$ | `sum(x)` | nondecreasing | Affine |
| | $f(x) = (\|x_1\|, \ldots, \|x_n\|)$ | `abs(x)` | nondecreasing | Convex |
| | $f(x) = \max_i x_i$ | `max(x)` | nondecreasing | Convex |
| | $f(x) = \min_i x_i$ | `min(x)` | nondecreasing | Concave |
| | $f(x) = \max(x, 0)$ | `pos(x)` | nondecreasing | Convex |
| | $f(x) = \max(-x, 0)$ | `neg(x)` | nonincreasing | Convex |
| | $f(x) = \\|x\\|_1$ | `norm_1(x)` | nondecreasing $x \geq 0$<br>nonincreasing $x \leq 0$ | Convex |
| | $f(x) = \\|x\\|_\infty$ | `norm_inf(x)` | nondecreasing | Convex |
| Second-order cone atoms | | | | |
| | $f(x) = \\|x\\|_2$ | `norm_2(x)` | nondecreasing $x \geq 0$<br>nonincreasing $x \leq 0$ | Convex |
| | $f(X) = \\|X\\|_F$ | `norm_fro(X)` | nondecreasing $X \geq 0$<br>nonincreasing $X \leq 0$ | Convex |
| | $f(x) = x^2$ | `square(x)` | nondecreasing $x \geq 0$<br>nonincreasing $x \geq 0$ | Convex |
| | $f(x) = \sqrt{x}$ | `sqrt(x)` | nondecreasing $x \geq 0$ | Concave |
| | $f(x,y) = \sqrt{xy}$ | `geo_mean(x,y)` | nondecreasing | Concave |
| | $f(x,y) = x^T x / y$ | `quad_over_lin(x,y)` | nonincreasing in $y$ for $y > 0$<br>nondecreasing in $x$ for $x \geq 0$<br>nonincreasing in $x$ for $x \leq 0$ | Convex |
| | $f(x) = 1/x$ | `inv_pos(x)` | nonincreasing | Convex |
| | $f(x) = \sum_i x_i^2$ | `sum_squares` | nondecreasing in $x$ for $x \geq 0$<br>nonincreasing in $x$ for $x \leq 0$ | Convex |

**Table 4: Conic Form Problem Solvers**

| Name | Language | LP | SOCP | SDP | Exp | Method | Open source |
|---|---|---|---|---|---|---|---|
| CLP [42] | C++ | X | | | | Simplex | X |
| Gurobi [40] | | X | X | | | Simplex, Interior Point, ... | |
| GLPK [31] | C | X | | | | Simplex, Interior Point, ... | X |
| MOSEK [33] | | X | X | X | | Simplex, Interior Point, ... | |
| ECOS [14] | C | X | X | X | | Interior Point | X |
| SCS [39] | C | X | X | X | X | Primal-Dual Operator Splitting | X |

for enterprise-wide dynamic optimization. *Computers & Chemical Engineering*, 33(3):575–582, 2009.

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

[8] A. Brooke, D. Kendrick, A. Meeraus, and R. Raman. The general algebraic modeling system. *GAMS Development Corporation*, 1998.

[9] M. R. Bussieck and A. Meeraus. General algebraic modeling system (GAMS). In *Modeling languages in mathematical optimization*, pages 137–157. Springer, 2004.

[10] E. Chu, N. Parikh, A. Domahidi, and S. Boyd. Code generation for embedded second-order cone programming. In *Control Conference (ECC), 2013 European*, pages 1547–1552. IEEE, 2013.

[11] CPLEX, IBM ILOG. V12. 1: UserâĂŹs manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009.

[12] C. Crusius. *Automated analysis of convexity properties of nonlinear programs*. PhD thesis, Stanford University, 2003.

[13] S. Diamond, E. Chu, and S. Boyd. CVXPY: A Python-embedded modeling language for convex optimization, version 0.2. `http://cvxpy.org/`, May 2014.

[14] A. Domahidi, E. Chu, and S. Boyd. ECOS: An SOCP solver for embedded systems. In *European Control Converence*, Zurich, Switzerland, 2013.

[15] R. Fourer, D. Gay, and B. Kernighan. *AMPL*. Boyd & Fraser, 1993.

[16] 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*, 22:26–43, 2010.

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

[18] M. Grant and S. 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, 2008.

[19] M. Grant, S. Boyd, and Y. Ye. Disciplined convex programming. In L. Liberti and N. Maculan, editors, *Global Optimization: From Theory to Implementation*, Nonconvex Optimization and its Applications, pages 155–210. Springer, 2006.

[20] M. Grant, S. Boyd, and Y. Ye. CVX: Matlab software for disciplined convex programming, ver. 2.0, build 870. `http://cvxr.com`, Sept. 2012.

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

[22] B. Houska, H. Ferreau, and M. Diehl. ACADO Toolkit – An open source framework for automatic control and dynamic optimization. *Optimal Control Applications and Methods*, 32(3):298–312, 2011.

[23] S. G. Johnson. The NLopt nonlinear-optimization package, 2010.

[24] L. V. Kantorovich. On a mathematical symbolism convenient for performing machine calculations. *Dokl. Akad. Nauk SSSR*, 113(4):738–741, 1957.

[25] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311. ACM, 1984.

[26] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[27] T. Lipp and S. Boyd. Variations and extensions of the convex-concave procedure. 2014.

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

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

[30] M. Lubin and I. Dunning. Computing in operations research using julia. *arXiv preprint arXiv:1312.1431*, 2013.

[31] A. Makhorin. GLPK (GNU linear programming kit), 2008.

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

[33] A. Mosek. The MOSEK optimization software. 54, 2010.

[34] A. S. Nemirovsky and D. B. Yudin. *Informational Complexity and Efficient Methods for Solution of Convex Extremal Problems*. Wiley, New York, 1983.

[35] 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.

[36] Y. Nesterov. Interior-point methods: An old and new approach to nonlinear programming. *Mathematical Programming*, 79(1):285–297, 1997.

[37] Y. Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer, 2004.

[38] Y. Nesterov and A. Nemirovskii. *Interior-point polynomial algorithms in convex programming*, volume 13. SIAM, 1994.

[39] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. Operator splitting for conic optimization via homogeneous self-dual embedding. *arXiv preprint arXiv:1312.3039*, 2013.

[40] G. Optimization. Gurobi optimizer reference manual. 2012.

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

[42] The Computational Infrastructure for Operations Research project. COIN-OR linear program solver. 2014.

[43] T. Van Voorhis and F. A. Al-Khayyal. Difference of

convex solution of quadratically constrained optimization problems. *European Journal of Operational Research*, 148(2):349–362, 2003.

[44] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM review*, 38(1):49–95, 1996.

[45] H. Wolkowicz, R. Saigal, and L. Vandenberghe. *Handbook of semidefinite programming: theory, algorithms, and applications*, volume 27. Springer, 2000.

[46] 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.

## APPENDIX

## A. SPEED TESTS: CODE

### A.1 Summation

Convex.

```
n = 10000
@time begin
x = Variable();
e = 0;
for i = 1:n
  e = e + x;
end
p = minimize(norm2(e-1), x >= 0);
solve!(p, ECOS.ECOSMathProgModel())
end
```

*CVX.*

```
n = 10000;
tic()
cvx_begin
variable x
e = 0
for i=1:n
    e = e + x;
end
minimize norm(e-1, 2)
subject to
    x >= 0;
cvx_end
toc()
```

*CVXPY.*

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

### A.2 Indexing

Convex.

```
n = 10000
@time begin
x = Variable(n);
e = 0;
for i = 1:n
  e = e + x[i];
end
p = minimize(norm2(e-1), x >= 0);
solve!(p, ECOS.ECOSMathProgModel())
end
```

*CVX.*

```
n = 10000;
tic()
cvx_begin
variable x(n)
e = 0;
for i=1:n
    e = e + x(i);
end
minimize norm(e - 1, 2)
subject to
    x >= 0;
cvx_end
toc()
```

*CVXPY.*

```
%%timeit
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.solve("ECOS", verbose=True)
```

### A.3 Transpose

Convex.

```
n = 500
A = randn(n, n);
@time begin
X = Variable(n, n);
p = minimize(vecnorm(X' - A), X[1,1] == 1);
solve!(p, ECOS.ECOSMathProgModel())
end
```

*CVX.*

```
n = 500;
A = randn(n, n);
tic()
cvx_begin
variable X(n, n);
minimize(norm(transpose(X) - A, 'fro'))
subject to
    X(1,1) == 1;
cvx_end
toc()
```

*CVXPY.*

```
n = 500
A = numpy.random.randn(n,n)
%%timeit
X = Variable(n,n)
p = Problem(Minimize(norm(X.T-A,'fro')), [X[1,1] == 1])
p.solve("ECOS", verbose=True)
```

## A.4 Matrix constraint

```
Convex.
```

```
n = 500;
A = randn(n, n);
B = randn(n, n);
@time begin
X = Variable(n, n);
p = minimize(vecnorm(X - A), X == B);
solve!(p, ECOS.ECOSMathProgModel())
end
```

*CVX.*

```
n = 500;
A = randn(n, n);
B = randn(n, n);
tic()
cvx_begin
variable X(n, n);
minimize(norm(X - A, 'fro'))
subject to
    X == B;
cvx_end
toc()
```

*CVXPY.*

```
n = 500
A = numpy.random.randn(n,n)
B = numpy.random.randn(n,n)
%%timeit
X = Variable(n,n)
p = Problem(Minimize(norm(X-A,'fro')), [X == B])
p.solve(verbose=True)
```